```python
import copy

import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm


class Dataset(object):
    """
    class for handling data
    """


    def __init__(self, path: str) -> None:
        """
        initialize params for handling data
        Args:
            path (str): path to data
        """
        self.k, self.char_to_ind, self.ind_to_char, self.book_data = Dataset._read_data(path)
        self.seq_x = None
        self.seq_y = None

    def set_seq_data(self, e: int=0, seq_length: int=25) -> None:
        """
        sets a sequence for x data and targets of seq_length
        Args:
            e (int): start of sequence. Defaults to 0.
            seq_length (int): length of sequence to evaluate. Defaults to 25.
        """
        X_chars = self.book_data[e : e+seq_length]
        Y_chars = self.book_data[e+1 : e+seq_length+1]
        self.seq_x = np.zeros(shape=(self.k, seq_length))
        self.seq_y = np.zeros(shape=(self.k, seq_length))
        for t in range(seq_length):
            self.seq_x[:,[t]] = Dataset._one_hot(char_i=self.char_to_ind[X_chars[t]], num_dist_chars=self.k)
            self.seq_y[:,[t]] = Dataset._one_hot(char_i=self.char_to_ind[Y_chars[t]], num_dist_chars=self.k)

    @staticmethod
    def _read_data(path: str) -> list((int, dict, dict, str)):
        """
        reads data from file and creates dictionarys mapping
        character <-> index
```

```
        Args:
            path (str): path to data

        Returns:
            list: list(
                K (int): length of unique chars
                char_to_ind (dict): map character to index
                ind_to_char (dict): map index to character
                book_data (str): all text data
            )
        """
        with open(path, 'r') as b:
            book_data = b.read()
        book_chars = set(book_data)
        K = len(book_chars)
        char_to_ind, ind_to_char = dict(), dict()
        for i, char in enumerate(book_chars):
            char_to_ind[char] = i
            ind_to_char[i] = char
        return K, char_to_ind, ind_to_char, book_data

    @staticmethod
    def _one_hot(char_i: int, num_dist_chars: int) -> np.array:
        """
        one-hot-encoding of char_i, size (num_dist_chars x 1)

        Args:
            char_i (int): index to encode
            num_dist_chars (int): length of vector

        Returns:
            np.array: one-hot-encoded vector
        """
        character_one_hot = np.zeros(shape=(num_dist_chars, 1))
        character_one_hot[char_i, 0] = 1
        return character_one_hot


class RNN(object):
    """
    class for handling params and methods of a RNN
    """



    def __init__(self, in_size: int, hidden_size: int, out_size: int, eta: float=0.1, seq_length: int=25, sig: float=0.01, rand_seed:
int=10) -> None:
        """
```

```
initialize params for RNN

Args:
    in_size (int): input length
    hidden_size (int): hidden layer length
    out_size (int): output length
    eta (float, optional): eta value when training network. Defaults to 0.1.
    seq_length (int, optional): length of sequence to evaluate. Defaults to 25.
    sig (float, optional): sigma value for initialization of node weights. Defaults to 0.1.
    rand_seed (int, optional): random seed for initialization of node weights. Defaults to 10.
"""
self.input_size = in_size
self.hidden_size = hidden_size
self.output_size = out_size
# randomize seed
np.random.seed(rand_seed)
# Initialization of weights and biases sizes (U: m x input_size, W: m x m, V: K x m, b: m x 1 c: K x 1)
self.U = np.random.normal(size=(self.hidden_size, self.input_size), loc=0, scale=sig)
self.W = np.random.normal(size=(self.hidden_size, self.hidden_size), loc=0, scale=sig)
self.V = np.random.normal(size=(self.output_size, self.hidden_size), loc=0, scale=sig)
self.b = np.zeros((self.hidden_size, 1))
self.c = np.zeros((self.output_size, 1))
# cache params for update of weights and biases
self.b_cache = np.zeros_like(self.b)
self.c_cache = np.zeros_like(self.c)
self.W_cache = np.zeros_like(self.W)
self.U_cache = np.zeros_like(self.U)
self.V_cache = np.zeros_like(self.V)
# store best model params
self.b_best = None
self.c_best = None
self.W_best = None
self.U_best = None
self.V_best = None
# misc params
self.seq_len = seq_length
self.eta = eta
self.smooth_loss = list()
self.update_num = 0
self.min_loss = 10000.0
self.opt_update_num = None
self.h_prev = None
```

```python
    def _set_seq_len(self, seq_len: int) -> None:
        """
        sets the seq length of the RNN to seq_len
        Args:
            seq_len (int): new seq length
        """
        self.seq_len = seq_len

    def _set_opt_param(self) -> None:
        """
        store best params
        """
        self.b_best = copy.deepcopy(self.b)
        self.c_best = copy.deepcopy(self.c)
        self.W_best = copy.deepcopy(self.W)
        self.U_best = copy.deepcopy(self.U)
        self.V_best = copy.deepcopy(self.V)


    @staticmethod
    def _tanh(a: np.ndarray) -> np.ndarray:
        """
        hyperbolic tangent function, implemented with numpy.tanh()
        Args:
            a (np.ndarray): vector to perform hyperbolic tangent function on
        Returns:
            np.ndarray: vector after hyperbolic tangent function
        """
        return np.tanh(a)

    @staticmethod
    def _softmax(x: np.ndarray) -> np.ndarray:
        """
        function that converts a vector of K real numbers into a
        probability distribution of K possible outcomes
        Args:
            x (np.array): vector to perform softmax on
        Returns:
            np.ndarray: vector after softmax
        """
        exp_x = np.exp(x - np.max(x))
        return exp_x / np.sum(exp_x)
```

```python
    @staticmethod
    def _gradient_clip(grad: dict) -> dict:
        """
        function to prevent exploding gradients
        matlab-func from lab instructions converted to python

        Args:
            grad (dict): dictionary of gradients for RNN weights and biases

        Returns:
            dict: dictionary of gradients for RNN weights and biases clipped
        """
        for key in grad.keys():
            grad[key] = np.clip(grad[key], -5, 5)
        return grad

    def syntesized_to_file(self, y: np.ndarray, ind_to_char: dict, update: int, loss: float) -> str:
        """
        maps one-hot-encoded input to string using mapping dict

        Args:
            y (np.ndarray): one-hot-encoded input, output from self.synthesize
            ind_to_char (dict): dictonary mapping index to character

        Returns:
            str: string of synthesized text
        """
        y_argmax = np.argmax(y, axis=0)
        txt_out = str()
        for i in y_argmax:
            txt_out += ind_to_char[i]

        tfile = open('synthesized_potter3.txt', 'a')
        tfile.write(f"update={update}, loss={loss}")
        tfile.write(f"\n")
        tfile.write(f"{txt_out}")
        tfile.write(f"\n")
        tfile.write(f"\n")
        tfile.close()

    def evaluate_classifier(self, h: np.ndarray, x: np.ndarray, best_params: bool=False) -> list((np.ndarray, np.ndarray, np.ndarray,
np.ndarray)):
        """
        evaluate a sequence, implements eq. 1-4 from lab instructions

        Args:
            h (np.ndarray): the hidden state at time t
            x (np.ndarray): the first (dummy) input vector
```

```python
            best_params (bool): True if evaluate with best params else False
        Returns:
            list: list(
                a (np.ndarray): output vector - in-to-hidden
                h (np.ndarray): output vector from hyperbolic tangent function on (a)
                o (np.ndarray): output vector - hidden-to-out
                p (np.ndarray): output vector from softmax on (o)
            )
        """
        b = self.b if not best_params else self.b_best
        c = self.c if not best_params else self.c_best
        W = self.W if not best_params else self.W_best
        U = self.U if not best_params else self.U_best
        V = self.V if not best_params else self.V_best
        a = np.matmul(W, h) + np.matmul(U, x) + b
        h = RNN._tanh(a)
        o = np.matmul(V, h) + c
        p = RNN._softmax(o)
        return a, h, o, p

    def synthesize(self, h0: np.ndarray, x0: np.ndarray, n: int, best_params: bool=False) -> np.ndarray:
        """
        synthesize a sequence of characters
        Args:
            h0 (np.ndarray): the hidden state at time t=0
            x0 (np.ndarray): the first (dummy) input vector
            n (int): length of sequence
            best_params (bool): True if evaluate with best params else False
        Returns:
            np.ndarray: output length of synthesize a sequence, one-hot-encoded
        """
        Y = np.zeros((self.output_size, n))
        x_t = x0
        h_tmin1 = h0
        for t in range(n):
            _, h_t, _, p_t = self.evaluate_classifier(h=h_tmin1, x=x_t, best_params=best_params)
            x_t = np.random.multinomial(1, np.squeeze(p_t))[:,np.newaxis]
            Y[:,[t]] = x_t[:,[0]]
            h_tmin1 = h_t
        return Y

    def forward_pass(self, x: np.ndarray, y: np.ndarray, h0: np.ndarray)  -> list((float, np.ndarray, np.ndarray, np.ndarray)):
        """
```

```python
        calculates the loss of the RNN
        Args:
            x (np.ndarray): sequence of x data, one-hot-encoded
            y (np.ndarray): sequence of target data, one-hot-encoded
            h0 (np.ndarray): the hidden state at time t=0
        Returns:
            list: list(
                loss (float): loss of the rnn
                p: probabilities vector
                h: hidden states vector
                a: output vector - in-to-hidden
            )
        """
        p, o, h, a = [None] * self.seq_len, [None] * self.seq_len, [None] * self.seq_len, [None] * self.seq_len
        h_tmin1 = h0
        loss = 0
        for t in range(self.seq_len):
            a[t], h[t], o[t], p[t] = self.evaluate_classifier(h=h_tmin1, x=x[:,[t]])
            loss -= np.log(np.matmul(y[:,[t]].T, p[t]))[0,0]
            h_tmin1 = h[t]
        return loss, p, [h0] + h, a

    def backward_pass(self, x: np.ndarray, y: np.ndarray, p: np.ndarray, h: np.ndarray) -> dict:
        """
        calculates the gradients of weights and biases layers
        Args:
            x (np.ndarray): sequence of x data, one-hot-encoded
            y (np.ndarray): sequence of target data, one-hot-encoded
            p (np.ndarray): probabilities vector
            h (np.ndarray): hidden states vector
        Returns:
            dict: dict(
                'b': biases gradients, in
                'c': biases gradients, out
                'W': weight gradients, in
                'U': weight gradients, hidden
                'V': weight gradients, out
            )
        """
        h0 = h[0]
        h = h[1:]
        grads = {
            'b': np.zeros_like(self.b),
```

```
            'c': np.zeros_like(self.c),
            'W': np.zeros_like(self.W),
            'U': np.zeros_like(self.U),
            'V': np.zeros_like(self.V),
        }
        grad_a = [None] * self.seq_len
        for t in range((self.seq_len-1), -1, -1):
            g = -(y[:,[t]] - p[t]).T
            grads['V'] += np.matmul(g.T, h[t].T)
            grads['c'] += g.T
            if t < (self.seq_len-1):
                dL_h = np.matmul(g, self.V)  + np.matmul(grad_a[t+1], self.W)
            else:
                dL_h = np.matmul(g, self.V)
            grad_a[t] = np.matmul(dL_h, np.diag(1 - h[t][:, 0]**2))
            if t==0:
                grads['W'] += np.matmul(grad_a[t].T, h0.T)
            else:
                grads['W'] += np.matmul(grad_a[t].T, h[t-1].T)
            grads['U'] += np.matmul(grad_a[t].T, x[:,[t]].T)
            grads['b'] += grad_a[t].T
            grads = RNN._gradient_clip(grads)
        return grads

    def update_params(self, grads: dict, eps: float=np.finfo(float).eps) -> None:
        """
        update rnn params according to Adagrad optimizer
        Args:
            grads (dict): dict from backward_pass backprop containing new param gradients
            eps (float): small number. Default to np.finfo(float).eps.
        """
        for key in grads.keys():
            vars(self)[key+'_cache'] += grads[key]**2
            vars(self)[key] += -self.eta * (grads[key] / (np.sqrt(vars(self)[key+'_cache']) + eps))

    def fit(self, data: Dataset, epochs: int=3, seq_len: int=25) -> None:
        """
        fit the RNN to the data, divides data into batches of seq-len length
        Args:
            data (Dataset): Dataset obj
            epochs (int): number of epochs to run. Defaults to 0.1.
            seq_len (int): length of mini batch
        """
        if seq_len != self.seq_len:
```

```python
            self.set_seq_len(seq_len)

        for epoch in tqdm(range(epochs)):
            h_prev = np.zeros(shape=(self.hidden_size, 1))
            for e in range(0, len(data.book_data)-1, self.seq_len):
                if e > len(data.book_data) - (seq_len-1):
                    print(f"epoch {epoch+1}, done")
                    break
                data.set_seq_data(e=e, seq_length=self.seq_len)

                loss, p, h, _ = self.forward_pass(data.seq_x, data.seq_y, h_prev)
                grads = self.backward_pass(data.seq_x, data.seq_y, p, h)

                if self.update_num == 0:
                    self.smooth_loss.append(loss)
                else:
                    self.smooth_loss.append(0.999 * self.smooth_loss[-1] + 0.001 * loss)

                if self.smooth_loss[-1] < self.min_loss:
                    self.min_loss = self.smooth_loss[-1]
                    self.opt_update_num = self.update_num
                    self._set_opt_param()

                self.update_params(grads)
                self.h_prev = h_prev = h[-1]

                if self.update_num % 10000 == 0 or self.update_num == 0:
                    Y = self.synthesize(h0=h_prev, x0=data.seq_x[:, [0]], n=200)
                    self.syntesized_to_file(y=Y, ind_to_char=data.ind_to_char, update=self.update_num, loss=self.smooth_loss[-1])

                self.update_num += 1


def ComputeGradsNum(rnn: RNN, x: np.ndarray, y: np.ndarray, h0: np.ndarray, h: float=1e-4) -> dict:
    """
    calculates numerical gradients of the RNN for sanity checking and correctness
    matlab-func from lab instructions converted to python
    Args:
        rnn (RNN): RNN obj
        x (np.ndarray): sequence of x data, one-hot-encoded
        y (np.ndarray): sequence of target data, one-hot-encoded
        h0 (np.ndarray): the hidden state at time t=0
        h (float, optional): small float number. Defaults to 1e-4.
```

```
    Returns:
        dict: dict(
            'b': biases gradients, in
            'c': biases gradients, out
            'W': weight gradients, in
            'U': weight gradients, hidden
            'V': weight gradients, out
        )
    """
    grads = {
        'b': np.zeros_like(rnn.b),
        'c': np.zeros_like(rnn.c),
        'W': np.zeros_like(rnn.W),
        'U': np.zeros_like(rnn.U),
        'V': np.zeros_like(rnn.V),
    }
    for key in tqdm(grads.keys()):
        for i in range(grads[key].shape[0]):
            for j in range(grads[key].shape[1]):
                rnn_try = copy.deepcopy(rnn)
                vars(rnn_try)[key][i, j] += h
                loss2 = rnn_try.forward_pass(x, y, h0)[0]
                vars(rnn_try)[key][i, j] -= 2 * h
                loss1 = rnn_try.forward_pass(x, y, h0)[0]
                grads[key][i, j] = (loss2 - loss1) / (2 * h)
    return grads


def plot_loss_curve(smooth_loss, title='', length_text=None, seq_length=None):
    """
    plots a loss curve
    """
    _, ax = plt.subplots(1, 1, figsize=(15,5))
    ax.plot(range(1, len(smooth_loss)+1), smooth_loss)

    # Add axis, legend and grid
    ax.set_xlabel('Update step')
    ax.set_ylabel('Smooth Loss')
    #ax.legend()
    #ax.grid(True)
    plt.show()
```

```python
if __name__ == "__main__":
    #data = Dataset('data/goblet_book.txt')

    """ # gradient testing
    data.set_seq_data(e=0, seq_length=25)
    rnn = RNN(in_size=data.k, hidden_size=5, out_size=data.k)
    h0 = np.zeros(shape=(rnn.hidden_size, 1))
    loss, p, h, a = rnn.forward_pass(data.seq_x, data.seq_y, h0)
    new_grads = rnn.backward_pass(data.seq_x, data.seq_y, p, h)
    new_grads_num = ComputeGradsNum(rnn, data.seq_x, data.seq_y, h0)
    for parameter in ['b','c','U','W','V']:
        abs_error = abs(new_grads_num[parameter] - new_grads[parameter])
        rel_error = abs(new_grads_num[parameter] - new_grads[parameter]) / (new_grads_num[parameter] + 1e-18)
        rel_error_max = rel_error.max()
        mean_abs_error_ = np.mean(abs_error, axis=0)
        mean_abs_error = np.mean(abs_error)
        print('For '+parameter+', the maximum relative error is '+str(rel_error_max)+ \
            ' and the mean absolute error is '+str(mean_abs_error)) """


    """ # training and plotting
    rnn = RNN(in_size=data.k, hidden_size=100, out_size=data.k)
    rnn.fit(data=data)
    synth_seq = rnn.synthesize(h0=rnn.h_prev, x0=data.seq_x[:, [0]], n=1000, best_params=True)
    rnn.syntesized_to_file(y=synth_seq, ind_to_char=data.ind_to_char, update=rnn.opt_update_num, loss=rnn.min_loss)
    plot_loss_curve(rnn.smooth_loss) """
```