Recurrent Neural Networks

Inclass Project 3 - MA4144

This project contains multiple tasks to be completed, some require written answers. Open a markdown cell below the respective question that require written answers and provide (type) your answers. Questions that required written answers are given in blue fonts. Almost all written questions are open ended, they do not have a correct or wrong answer. You are free to give your opinions, but please provide related answers within the context.

After finishing project run the entire notebook once and **save the notebook as a pdf** (File menu -> Save and Export Notebook As -> PDF). You are **required to upload both this ipynb file and the PDF on moodle**.

Outline of the project

The aim of the project is to build a RNN model to suggest autocompletion of half typed words. You may have seen this in many day today applications; typing an email, a text message etc. For example, suppose you type in the four letter "univ", the application may suggest you to autocomplete it by "university".

We will train a RNN to suggest possible autocompletes given 3 - 4 starting letters. That is if we input a string "univ" hopefully we expect to see an output like "university", "universal" etc.

For this we will use a text file (wordlist.txt) containing 10,000 common English words (you'll find the file on the moodle link). The list of words will be the "**vocabulary**" for our model.

We will use the Python **torch library** to implement our autocomplete model.

Use the below cell to use any include any imports

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import random
from tqdm import tqdm
```

Section 1: Preparing the vocabulary

```
WORD_SIZE = 13
```

Q1. In the following cell provide code to load the text file (each word is in a newline), then extract the words (in lowercase) into a list.

For practical reasons of training the model we will only use words that are longer that 3 letters and that have a maximum length of WORD_SIZE (this will be a constant we set at the beginning - you can change this and experiment with different WORD_SIZEs). As seen above it is set to 13.

So out of the extracted list of words filter out those words that match our criteria on word length.

To train our model it is convenient to have words/strings of equal length. We will choose to convert every word to length of WORD_SIZE, by adding underscores to the end of the word if it is initially shorter than WORD_SIZE. For example, we will convert the word "university" (word length 10) into "university___" (wordlength 13). In your code include this conversion as well.

Store the processed WORD_SIZE lengthed strings in a list called vocab.

```
vocab = []

with open("wordlist.txt", "r") as file:
    for line in file:
        word = line.strip().lower()
        if 4 <= len(word) <= WORD_SIZE:
            padded_word = word.ljust(WORD_SIZE, '_')
            vocab.append(padded_word)

# print(vocab[:10])
```

In the above explanation it was mentioned "for practical reasons of training the model we will only use words that are longer that 3 letters and that have a certain maximum length". In your opinion what could be those practical? Will hit help to build a better model?

Answer (to write answers edit this cell)

1. Very short words does not contain enough information and also they do not need to be predicted since there are less than 3 number of characters. And the ambigiouty is high when the model tries to predict words by looking at only 1 or 2 characters.

2. Fixed-length sequences make batching and training much more efficient.

3. To have too long words are also unnecessary since the use case of those words are very limited compared to the common words. Therefore using computation ppower to train the model for such rare cases are not necassary since they take both memory and time.

Q2 To input words into the model, we will need to convert each letter/character into a number. as we have seen above, the only characters in our list vocab will be the underscore and lowercase english letters. so we will convert these 27 characters into numbers as follows: underscore -> 0, 'a' -> 1, 'b' -> 2, ···, 'z' -> 26. In the following cell,

(i) Implement a method called char_to_num, that takes in a valid character and outputs its numerical assignment.

(ii) Implement a method called num_to_char, that takes in a valid number from 0 to 26 and outputs the corresponding character.

(iii) Implement a method called word_to_numlist, that takes in a word from our vocabulary and outputs a (torch) tensor of numbers that corresponds to each character in the word in that order. For example: the word "united_____" will be converted to tensor([21, 14, 9, 20, 5, 4, 0, 0, 0, 0, 0, 0, 0]). You are encouraged to use your char_to_num method for this.

(iv) Implement a method called numlist_to_word, that does the opposite of the above described word_to_numlist, given a tensor of numbers from 0 to 26, outputs the corresponding word. You are encouraged to use your num_to_char method for this.

Note: As mentioned since we are using the torch library we will be using tensors instead of the usual python lists or numpy arrays. Tensors are the list equivalent in torch. Torch models only accept tensors as input and they output tensors.

```
def char_to_num(char):

    if char == '_':
        return 0

    num = ord(char) - ord('a') + 1
    return num


def num_to_char(num):

    if num == 0:
        return '_'

    char = chr(num + ord('a') - 1)
    return(char)


def word_to_numlist(word):
```

```
    num_list = [char_to_num(c) for c in word]
    numlist = torch.tensor(num_list, dtype=torch.long)
    return(numlist)


def numlist_to_word(numlist):

    chars = [num_to_char(num.item()) for num in numlist]
    word =  ''.join(chars)
    return(word)
```

We convert letter into just numbers based on their aphabetical order, I claim that it is a very bad way to encode data such as letters to be fed into learning models, please write your explanation to or against my claim. If you are searching for reasons, the keyword 'categorical data' may be useful. Although the letters in our case are not treated as categorical data, the same reasons as for categorical data is applicable. Even if my claim is valid, at the end it won't matter due to something called "embedding layers" that we will use in our model. What is an embedding layer? What is it's purpose? Explain.

Answer (to write answers edit this cell)

The suggested claim is right because of following reasons

1. It imposes a false ordinal relationship between characters.
2. Characters are categorical, and numerical distance between indices can mislead the model into learning non-existent similarities.
3. The model may learn spurious patterns due to the artificial ordering of characters.

Embedding layer is a trainable lookup table that maps each integer to a dense vector of real numbers.

Purpose of embedding layer :

1. Converts discrete character IDs into continuous-valued vectors.
2. Learns semantic relationships.
3. Avoids sparse one-hot vectors, reducing memory usage and improving learning efficiency.

Section 2: Implementing the Autocomplete model

We will implement a RNN LSTM model. The video tutorial will be useful. Our model will be only one hidden layer, but feel free to sophisticate with more layers after the project for your own experiments.

Our model will contain all the training and prediction methods as single package in a class (autocompleteModel) we will define and implement below.

```
LEARNING_RATE = 0.005
```

```
class autocompleteModel(nn.Module):

    #Constructor
    def __init__(self, alphabet_size, embed_dim, hidden_size,
num_layers):
        super().__init__()

        self.alphabet_size = alphabet_size
        self.embed_dim = embed_dim
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        #TODO

        #Initialize the layers in the model:
        #1 embedding layer, 1 - LSTM cell (hidden layer), 1 fully
connected layer with linear activation
        self.embed_layer = nn.Embedding(alphabet_size, embed_dim)
        self.LSTM_cell = nn.LSTMCell(embed_dim, hidden_size)
        self.fc = nn.Linear(hidden_size, alphabet_size)


    #Feedforward
    def forward(self, character, hidden_state, cell_state):

        #Perform feedforward in order
        #1. Embed the input (one charcter represented by a number)
        embedding = self.embed_layer(character)

        #2. Feed the embedded output to the LSTM cell
        hidden_state, cell_state = self.LSTM_cell(embedding,
(hidden_state, cell_state))

        #3. Feed the LSTM output to the fully connected layer to
obtain the output
        output = self.fc(hidden_state)

        #4. return the output, and both the hidden state and cell
state from the LSTM cell output
        return output, hidden_state, cell_state

    #Intialize the first hidden state and cell state (for the start of
a word) as zero tensors of required length.
    def initial_state(self, batch_size=1):
        h0 = torch.zeros(1, self.hidden_size)
        c0 = torch.zeros(1, self.hidden_size)

        return (h0, c0)

    #Train the model in epochs given the vocab, the training will be
```

```
fed in batches of batch_size
    def trainModel(self, vocab, epochs = 5, batch_size = 100,
learning_rate = LEARNING_RATE, plot_status = False):

        #Convert the model into train mode
        self.train()

        #Set the optimizer (ADAM), you may need to provide the model
parameters  and learning rate
        optimizer = optim.Adam(self.parameters(), learning_rate)

        #Keep a log of the loss at the end of each training cycle.
        loss_log = []

        for e in range(epochs):

             #TODO: Shuffle the vocab list the start of each epoch
            random.shuffle(vocab)

            num_iter = len(vocab) // batch_size

            batch_loss_list = []

            accuracy = 0.0

            for i in tqdm(range(num_iter)):

                #TODO: Set the loss to zero, initialize the optimizer
with zero_grad at the beginning of each training cycle.
                batch_loss = 0
                optimizer.zero_grad()

                vocab_batch = vocab[i * batch_size:(i + 1) *
batch_size]

                for word in vocab_batch:

                    # Initialize the hidden state and cell state at
the start of each word.
                    hidden_state,cell_state = self.initial_state()

                    # Convert the word into a tensor of number and
create input and target from the word
                    #Input will be the first WORD_SIZE - 1 charcters
and target is the last WORD_SIZE - 1 charcters
                    input_tensor = word_to_numlist(word[:WORD_SIZE-1])
                    target_tensor = word_to_numlist(word[1:WORD_SIZE])

                    #Loop through each character (as a number) in the
word
                    for c in range(WORD_SIZE - 1):
```

```
                        #TODO: Feed the cth character to the model
(feedforward) and comput the loss (use cross entropy in torch)
                        output, hidden_state, cell_state =
self.forward(input_tensor[c].unsqueeze(0), hidden_state, cell_state)
                        loss =
nn.functional.cross_entropy(output,target_tensor[c].view(1))
                        batch_loss += loss
                        accuracy += (output.argmax(dim=1) ==
target_tensor[c]).sum().item()


                #TODO: Compute the average loss per word in the batch
and perform backpropagation (.backward())
                batch_loss /= len(vocab_batch)
                batch_loss.backward()

                #TODO: Update model parameters using the optimizer
                optimizer.step()

                #Update the loss_log
                batch_loss_list.append(batch_loss.item())

            accuracy /= (13*batch_size*num_iter)

            print("Epoch: ", e+1, " Training loss per word : ",
batch_loss.item(), f" Training accuracy: {accuracy*100} % ")

            loss_log.append(np.mean(batch_loss_list))

        loss_log_processed = np.array(loss_log, dtype=np.float16)

        if plot_status:
            # Plot a graph of the variation of the loss.
            plt.figure(figsize=(6,4))
            plt.plot(loss_log_processed)
            plt.xlabel('Epochs')
            plt.ylabel('Loss')
            plt.title('Training Loss vs Epochs')
            plt.show()

        return accuracy

    #Perform autocmplete given a sample of strings (typically 3-5
starting letters)
    def autocomplete(self, sample):

        #Convert the model into evaluation mode
        self.eval()
        completed_list = []

        #TODO: In the following loop for each sample item initialize
```

```
hidden and cell states, then predict the remaining characters
        #You will have to convert the output into a softmax (you may
use your softmax method from the last project) probability
distribution, then use torch.multinomial
        for i in tqdm(range(len(sample))):

            literal = sample[i]

            hidden_state,cell_state = self.initial_state()
            input_tensor = word_to_numlist(literal)
            predicted = literal

            for p in range(len(literal) - 1):
                init_input = input_tensor[p].unsqueeze(0)
                nl, hidden_state, cell_state =
self.forward(init_input, hidden_state, cell_state)

            init_input = input_tensor[-1].unsqueeze(0)

            for g in range(WORD_SIZE - len(literal)):
                # generate
                output, hidden_state, cell_state =
self.forward(init_input, hidden_state, cell_state)
                output_prob = nn.functional.softmax(output, dim = 1)
                top_1 = torch.multinomial(output_prob, 1)[0]

                # prediction
                pred_char = num_to_char(top_1)
                predicted += pred_char
                init_input =
torch.tensor(char_to_num(pred_char)).unsqueeze(0)

            completed_list.append(predicted)

        return(completed_list)
```

Section 3: Using and evaluating the model

(i) Initialize and train autocompleteModels using different embedding dimensions and hidden layer sizes. Use different learning rates, epochs, batch sizes. Train the best model you can.

(ii) Evaluate it on different samples of partially filled in words to test your model. Eg: ["univ", "math", "neur", "engin"] etc.

(iii) Set your best model, to the variable best_model. This model will be tested against random inputs (3-4 starting strings of common English words). **This will be the main contributor for your score in this project**.

```
import urllib.request

url = "https://raw.githubusercontent.com/SadeepRathnayaka/MA4144---
Neural-Network-and-Fuzzy-Logic/main/best_wieghts.pth"
save_path = "best_weights.pth"
urllib.request.urlretrieve(url, save_path)
```

```
('best_weights.pth', <http.client.HTTPMessage at 0x1b2c212cf30>)
```

```
best_model = None

model = autocompleteModel(alphabet_size=27, embed_dim=64,
hidden_size=256, num_layers=1)
model.load_state_dict(torch.load("best_weights.pth",
weights_only=False))  # Optional: explicitly set weights_only=False
model.eval()
```

```
autocompleteModel(
  (embed_layer): Embedding(27, 64)
  (LSTM_cell): LSTMCell(64, 256)
  (fc): Linear(in_features=256, out_features=27, bias=True)
)
```

```
word_seg = ["univ", "math", "neur", "engin","thi", "prod", "auto",
"reac", "stri"]
predictions = model.autocomplete(word_seg)

for a,prediction in zip(word_seg,predictions):
    pred_ = prediction.replace("_","")
    print(f'Input Word : {a} || Suggested Word : {pred_}')

# print(predictions)
```

```
100%|██████████| 9/9 [00:00<00:00, 376.15it/s]

Input Word : univ || Suggested Word : universal
Input Word : math || Suggested Word : mathemated
Input Word : neur || Suggested Word : neural
Input Word : engin || Suggested Word : engineers
Input Word : thi || Suggested Word : think
Input Word : prod || Suggested Word : produce
Input Word : auto || Suggested Word : automation
Input Word : reac || Suggested Word : reaction
Input Word : stri || Suggested Word : strikes
```

```
best_model = model
```