LAB 6 : Recurrent Neural Networks

Name :

Roll Number :

Referrence Material :

1. https://github.com/pangolulu/rnn-from-scratch
2. https://www.analyticsvidhya.com/blog/2019/01/fundamentals-deep-learning-recurrent-neural-networks-scratch-python/

Problem 1 : Next Token Prediction in a Sequence

Observation to be demonstrated:

1. Generate the data required
2. Represent tokens as indices using dictionaries
3. Convert the tokens into vectors using One hot encoding
4. Implement Recurrent Neural Network to solve the Next token prediction problem

Write down the Objectives, Hypothesis and Experimental description for the above problem

Double-click (or enter) to edit

Programming :

1. **Representing tokens or text**

In previous labs we mainly considered data $x \in \mathrm{R}^d$, where d is the feature space dimension. With time sequences our data can be represented as $x \in \mathrm{R}^{t \times d}$, where t is the sequence length. This emphasises sequence dependence and that the samples along the sequence are not independent and identically distributed (i.i.d.). We will model functions as $\mathrm{R}^{t \times d} \rightarrow \mathrm{R}^c$, where c is the amount of classes in the output.

There are several ways to represent sequences. With text, the challenge is how to represent a word as a feature vector in d dimensions, as we are required to represent text with decimal numbers in order to apply neural networks to it.

In this exercise we will use a simple one-hot encoding but for categorical variables that can take on many values (e.g. words in the English language) this may be infeasible. For such scenarios, you can project the encodings into a smaller space by use of embeddings.

2. **One-hot encoding over vocabulary**

One way to represent a fixed amount of words is by making a one-hot encoded vector, which consists of 0s in all cells with the exception of a single 1 in a cell used uniquely to identify each word.

vocabulary	one-hot encoded vector
Paris	$= [1, 0, 0, \ldots, 0]$
Rome	$= [0, 1, 0, \ldots, 0]$
Copenhagen	$= [0, 0, 1, \ldots, 0]$

Representing a large vocabulary with one-hot encodings often becomes inefficient because of the size of each sparse vector. To overcome this challenge it is common practice to truncate the vocabulary to contain the k most used words and represent the rest with a special symbol, `UNK`, to define unknown/unimportant words. This often causes entities such as names to be represented with `UNK` because they are rare.

Consider the following text

> I love the corny jokes in Spielberg's new movie.

where an example result would be similar to

> I love the corny jokes in `UNK`'s new movie.

Generating a dataset

For this exercise we will create a simple dataset that we can learn from. We generate sequences of the form:

`a a a a b b b b EOS, a a b b EOS, a a a a a b b b b b EOS`

where `EOS` is a special character denoting the end of a sequence. The task is to predict the next token t_n, i.e. `a`, `b`, `EOS` or the unknown token `UNK` given the sequence of tokens $\{t_1, t_2, \ldots, t_{n-1}\}$ and we are to process sequences in a sequential manner. As such, the network will need to learn that e.g. 5 `b`s and an `EOS` token will occur following 5 `a`s.

```
1 import numpy as np
```

```
 2
 3
 4 def generate_dataset(num_sequences=10):
 5     """
 6     Generates a number of sequences as our dataset.
 7
 8     Input :
 9      `num_sequences`: the number of sequences to be generated.
10
11     Returns a list of sequences.
12     """
13     samples = []
14
15     ## Write your code here
16
17
18     return samples
19
20
21 sequences = generate_dataset()
22
23 print('A single sample from the generated dataset:')
24 print(sequences[0])
```

```
A single sample from the generated dataset:
['a', 'a', 'a', 'b', 'b', 'b', 'EOS']
```

Representing tokens as indices

To build a one-hot encoding, we need to assign each possible word in our vocabulary an index. We do that by creating two dictionaries: one that allows us to go from a given word to its corresponding index in our vocabulary, and one for the reverse direction. Let's call them `word_to_idx` and `idx_to_word`. The keyword `num_words` specifies the maximum size of our vocabulary. If we try to access a word that does not exist in our vocabulary, it is automatically replaced by the `UNK` token or its corresponding index.

```
 1 # from collections import defaultdict
 2
 3 def sequences_to_dicts(sequences):
 4     """
 5     Create word_to_idx and idx_to_word dictionaries for a list of sequences.
 6     """
 7
 8     ## Write your code here
 9
10
11     return word_to_idx, idx_to_word, num_sequences, vocab_size
12
13
14 word_to_idx, idx_to_word, num_sequences, vocab_size = sequences_to_dicts(sequences)
15
```

```
print("Word to Index Dictionary : ",dict(word_to_idx))
print("Index to Word Dictionary : ",dict(idx_to_word))
print("Number of Sequences : ",num_sequences)
print("Vocab Size : ",vocab_size)
```

```
Word to Index Dictionary :  {'a': 0, 'b': 1, 'EOS': 2, 'UNK': 3}
Index to Word Dictionary :  {0: 'a', 1: 'b', 2: 'EOS', 3: 'UNK'}
Number of Sequences :  10
Vocab Size :  4
```

Creating Dataset

To build our dataset, we need to create inputs and targets for each sequences and partition sentences it into training and test sets. 80% and 20% is a common distribution, but mind you that this largely depends on the size of the dataset. **Since we are doing next-word predictions, our target sequence is simply the input sequence shifted by one word.**

```

def create_datasets(sequences, p_train=0.8, p_test=0.2):
    # Define partition sizes
    num_train = int(len(sequences)*p_train)
    num_test = int(len(sequences)*p_test)

    # Split sequences into partitions
    sequences_train = sequences[:num_train]
    sequences_test = sequences[-num_test:]

    def get_inputs_targets_from_sequences(sequences):
        # Define empty lists
        inputs, targets = [], []

        # Append inputs and targets s.t. both lists contain L-1 words of a sentence of
        # but targets are shifted right by one so that we can predict the next word

        ## Write your code here



        return inputs, targets

    # Get inputs and targets for each partition
    inputs_train, targets_train = get_inputs_targets_from_sequences(sequences_train)
    inputs_test, targets_test = get_inputs_targets_from_sequences(sequences_test)


    return inputs_train,targets_train,inputs_test,targets_test


x_train,y_train,x_test,y_test = create_datasets(sequences)

print("Input for the first training sample : ",x_train[0])
```

```
print("Target output for the first training sample : ",y_train[0])
```

```
Input for the first training sample :  ['a', 'a', 'a', 'b', 'b', 'b']
Target output for the first training sample :  ['a', 'a', 'b', 'b', 'b', 'EOS']
```

One-hot encodings

We now create a simple function that returns the one-hot encoded representation of a given index of a word in our vocabulary. Notice that the shape of the one-hot encoding is equal to the vocabulary (which can be huge!). Additionally, we define a function to automatically one-hot encode a sentence.

```
def one_hot_encode(idx, vocab_size):
    """
    One-hot encodes a single word given its index and the size of the vocabulary.

    Input :
     `idx`: the index of the given word
     `vocab_size`: the size of the vocabulary

    Returns a 1-D numpy array of length `vocab_size`.
    """
    # Initialize the encoded array
    one_hot = np.zeros(vocab_size)

    # Set the appropriate element to one
    one_hot[idx] = 1.0

    return one_hot


def one_hot_encode_sequence(sequence, vocab_size):
    """
    One-hot encodes a sequence of words given a fixed vocabulary size.

    Input :
     `sentence`: a list of words to encode
     `vocab_size`: the size of the vocabulary

    Returns a 3-D numpy array of shape (num words, vocab size, 1).
    """
    # Encode each word in the sentence
    encoding = np.array([one_hot_encode(word_to_idx[word], vocab_size) for word in sequ

    # Reshape encoding s.t. it has shape (num words, vocab size, 1)
    encoding = encoding.reshape(encoding.shape[0], encoding.shape[1], 1)

    return encoding


```

```
test_word = one_hot_encode(word_to_idx['a'], vocab_size)
print(f'Our one-hot encoding of \'a\' is {test_word}.')
print(f'Our one-hot encoding of \'a\' has shape {test_word.shape}.')

test_sentence = one_hot_encode_sequence(['a', 'b'], vocab_size)
print(f'Our one-hot encoding of \'a b\' is {test_sentence}.')
print(f'Our one-hot encoding of \'a b\' has shape {test_sentence.shape}.')
```

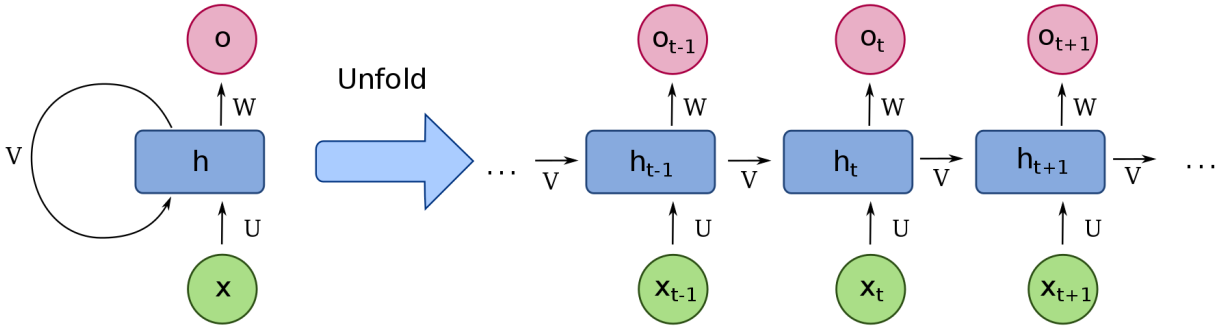
```
Our one-hot encoding of 'a' is [1. 0. 0. 0.].
Our one-hot encoding of 'a' has shape (4,).
Our one-hot encoding of 'a b' is [[[1.]
  [0.]
  [0.]
  [0.]]

 [[0.]
  [1.]
  [0.]
  [0.]]].
Our one-hot encoding of 'a b' has shape (2, 4, 1).
```

Implementation of RNN :

A recurrent neural network (RNN) is a type of neural network that has been succesful in modelling sequential data, e.g. language, speech, protein sequences, etc.

A RNN performs its computations in a cyclic manner, where the same computation is applied to every sample of a given sequence. The idea is that the network should be able to use the previous computations as some form of memory and apply this to future computations. An image may best explain how this is to be understood,



where it the network contains the following elements:

- x is the input sequence of samples,
- U is a weight matrix applied to the given input sample,

- V is a weight matrix used for the recurrent computation in order to pass memory along the sequence,
- W is a weight matrix used to compute the output of the every timestep (given that every

timestep requires an output),

- h is the hidden state (the network's memory) for a given time step, and
- o is the resulting output.

When the network is unrolled as shown, it is easier to refer to a timestep, t. We have the following computations through the network:

- $h_t = f(U\,x_t + V\,h_{t-1})$, where f usually is an activation function, e.g. $\tanh$.
- $o_t = \mathrm{softmax}(W\,h_t)$

Steps :

1. Implement Forward Pass, Backward Pass and Optimisation
2. Write the training loop
3. Take care of the exploding gradient problem by clipping the gradients

```
1 ## Write your code here
```

Inferences and Conclusion : State all the key observations and conclusion

Double-click (or enter) to edit

Problem 2 : Demonstrate the same for a Sine Wave

Objective : Given a sequence of 50 numbers belonging to a sine wave, predict the 51st number in the series.

Double-click (or enter) to edit

```
1 ## Write your code here
```

✓ 56s completed at 8:00 PM