01_Sequence_Processing_and_Language_Models

February 17, 2024

[1]:
```
import numpy as np
import torch
```

1 Sequential data

- In the following lectures, we will study neural networks that are able to make predictions based on an entire sequence of input vectors (or images) rather than a single vector (or image).
- We will focus on textual data, but the core ideas will generalize to other types of sequential data (such as time-series, video, and audio).
- A text can be represented by a sequence of characters (such as letters, numbers, and symbols). A text can also be represented by a sequence of pre-defined *tokens*, each of which is a sequence of characters.
- Neural networks currently achieve state-of-the-art performance in text classification, summarization, generation, and translation.

2 Text processing

- We will use the book Frankenstein; or, The Modern Prometheus by Mary Wollstonecraft Shelley to present a basic text processing pipeline.
- This pipeline will convert a text into a sequence of numbers, each of which corresponds to a token (sequence of characters).
- The following code downloads the book and stores it into a Python string. In order to simplify the implementation, sequences of non-letters are removed and uppercase letters are converted to lowercase letters.

[2]:
```
import requests
import re

def preprocess(text):
    text = re.sub('[^A-Za-z]+', ' ', text) # Substitutes any sequence of␣
 ↪non-letters by a whitespace
    text = text.lower() # Converts uppercase letters to lowercase letters
    return text

# Project Gutenberg has many books stored as text files (https://www.gutenberg.
↪org/)
raw_text = requests.get('https://www.gutenberg.org/cache/epub/84/pg84.txt').
↪text # Downloads and stores the text into a string
raw_text = raw_text.partition('*** START OF THE PROJECT GUTENBERG EBOOK␣
↪FRANKENSTEIN; OR, THE MODERN PROMETHEUS ***')[2] # Removes foreword
raw_text = raw_text.partition('*** END OF THE PROJECT GUTENBERG EBOOK␣
↪FRANKENSTEIN; OR, THE MODERN PROMETHEUS ***')[0] # Removes afterword

text = preprocess(raw_text)
```

[3]:
```
print(text[:150]) # Prints the first 150 characters
```

```
 frankenstein or the modern prometheus by mary wollstonecraft godwin shelley
contents letter letter letter letter chapter chapter chapter chapter chap
```

- A text can be represented by a sequence of pre-defined *tokens*, each of which is a sequence of characters. Different sets of tokens can lead to different text representations.
- For example, if each token is a word from a comprehensive list, “the modern prometheus” would be composed of 3 tokens (implicitly separated by whitespaces). If each token is a character, “the modern prometheus” would be composed of 21 tokens (including whitespaces).
- In this context, a vocabulary assigns an index (unique numerical identifier) to each token found in a text. Such indices can be one-hot encoded and provided to neural networks.
- In the code presented below, the function `tokenize` converts a text into a list of tokens (in this case, characters or sequences of letters) and the class `Vocabulary` implements a vocabulary.

[4]:
```
import collections

def tokenize(text, use_chars):
    if use_chars: # One token for each character
        return list(text)
    else: # One token for each sequence of letters
        return text.split()

class Vocabulary:
    def __init__(self, tokens):
        counter = collections.Counter(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[1],␣
↪reverse=True) # List of pairs where each pair is composed of a token and its␣
↪frequency. Sorted according to decreasing frequency.

        self.unknown = '?' # Represents an unknown token
        self.id_to_token = sorted([token for token, freq in self.token_freqs])␣
↪+ [self.unknown] # Maps an index to a token
        self.token_to_id = {token: id for id, token in enumerate(self.
↪id_to_token)} # Maps a token to an index

    def __len__(self):
        return len(self.id_to_token)

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)): # If called with a single␣
↪token
            return self.token_to_id.get(tokens, self.unknown)
        else: # If called with a list of tokens
            return [self.token_to_id.get(token, self.unknown) for token in␣
↪tokens]

    def to_tokens(self, indices):
        if not isinstance(indices, (list, tuple)): # If called with a single␣
↪index
            return self.id_to_token[indices]
        else: # If called with a list of indices
            return [self.id_to_token[index] for index in indices]
```

[5]:
```
use_chars = False # Choose between assigning one token for each character or␣
↪one token for each sequence of letters

tokens = tokenize(text, use_chars=use_chars)
print(tokens[:10])

vocab = Vocabulary(tokens)
print(vocab.id_to_token[:10])

indices = vocab[tokens]
print(indices[:10])

tokens_from_indices = vocab.to_tokens(indices)
print(tokens_from_indices[:10])
```

```
['frankenstein', 'or', 'the', 'modern', 'prometheus', 'by', 'mary',
'wollstonecraft', 'godwin', 'shelley']
['a', 'abandon', 'abandoned', 'abbey', 'abhor', 'abhorred', 'abhorrence',
'abhorrent', 'ability', 'abject']
[2608, 4298, 6169, 3991, 4811, 837, 3846, 6884, 2767, 5555]
['frankenstein', 'or', 'the', 'modern', 'prometheus', 'by', 'mary',
'wollstonecraft', 'godwin', 'shelley']
```

[6]:
```
print(vocab.token_freqs[:10])
```

```
[('the', 4195), ('and', 2976), ('i', 2850), ('of', 2642), ('to', 2094), ('my',
1776), ('a', 1391), ('in', 1129), ('was', 1021), ('that', 1018)]
```

- Sophisticated tokenization is implemented in widely available natural language processing libraries (such as spaCy). Such implementations employ language-specific tokenization rules and have pre-defined tokens extracted from large corpora (set of texts).
- The following code tokenizes the same raw text using spaCy.

[7]:
```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp(raw_text)
```

[8]:
```
spacy_tokens = [str(token) for token in doc]
print(spacy_tokens[:128])
```

```
['\r\n\r\n\r\n\r\n\r\n', 'Frankenstein', ';', '\r\n\r\n', 'or', ',', 'the',
'Modern', 'Prometheus', '\r\n\r\n', 'by', 'Mary', 'Wollstonecraft', '(',
'Godwin', ')', 'Shelley', '\r\n\r\n\r\n ', 'CONTENTS', '\r\n\r\n ', 'Letter',
'1', '\r\n ', 'Letter', '2', '\r\n ', 'Letter', '3', '\r\n ', 'Letter', '4',
'\r\n ', 'Chapter', '1', '\r\n ', 'Chapter', '2', '\r\n ', 'Chapter', '3', '\r\n
', 'Chapter', '4', '\r\n ', 'Chapter', '5', '\r\n ', 'Chapter', '6', '\r\n ',
'Chapter', '7', '\r\n ', 'Chapter', '8', '\r\n ', 'Chapter', '9', '\r\n ',
'Chapter', '10', '\r\n ', 'Chapter', '11', '\r\n ', 'Chapter', '12', '\r\n ',
'Chapter', '13', '\r\n ', 'Chapter', '14', '\r\n ', 'Chapter', '15', '\r\n ',
'Chapter', '16', '\r\n ', 'Chapter', '17', '\r\n ', 'Chapter', '18', '\r\n ',
'Chapter', '19', '\r\n ', 'Chapter', '20', '\r\n ', 'Chapter', '21', '\r\n ',
'Chapter', '22', '\r\n ', 'Chapter', '23', '\r\n ', 'Chapter', '24',
'\r\n\r\n\r\n\r\n\r\n', 'Letter', '1', '\r\n\r\n', '_', 'To', 'Mrs.', 'Saville',
',', 'England', '.', '_', '\r\n\r\n\r\n', 'St.', 'Petersburgh', ',', 'Dec.',
'11th', ',', '17-.', '\r\n\r\n\r\n', 'You', 'will', 'rejoice', 'to']
```

3 Language model

- A language model assigns a probability to each possible sequence of tokens (text).
- For example, consider the three following texts:
  1. “We are studying neural networks.”
  2. “Neural networks are studying us.”
  3. “Qwhuep zdmhxa qwerlj qwerz dsfa.”
- A good language model trained on a large corpora (such as the set of all webpages in English) would likely assign a relatively high probability to the first text, lower probability to the second text, and very low probability to the third text.
- Language models can be used to generate text, which in turn can be used to solve tasks defined by so-called prompts.

- For example, by generating text that starts with “I have a delicious recipe for a quick and healthy dinner:”, a language model can be used to suggest such a recipe.
- Language models also have other uses. For example, by comparing the probabilities of the texts “to recognize speech” and “to wreck a nice beach”, a language model can help a speech recognition system decide between these two interpretations of a speech recording.

4 Chain rule of probability

- Consider a sequence of discrete random variables $X_1, X_2, \dots, X_T$ and a sequence of values $x_1, x_2, \dots, x_T$ such that $\mathbb{P}(X_1 = x_1, X_2 = x_2, \dots, X_T = x_T) > 0$.
- The chain rule of probability states that

$$\mathbb{P}(X_1 = x_1, X_2 = x_2, \dots, X_T = x_T) = \mathbb{P}(X_1 = x_1) \prod_{t=2}^{T} \mathbb{P}(X_t = x_t \mid X_1 = x_1, \dots, X_{t-1} = x_{t-1}),$$

where the conditional probability $\mathbb{P}(X_t = x_t \mid X_1 = x_1, \dots, X_{t-1} = x_{t-1})$ is given by

$$\mathbb{P}(X_t = x_t \mid X_1 = x_1, \dots, X_{t-1} = x_{t-1}) = \frac{\mathbb{P}(X_1 = x_1, \dots, X_t = x_t)}{\mathbb{P}(X_1 = x_1, \dots, X_{t-1} = x_{t-1})}.$$

- For example, if $T = 3$,

$$\mathbb{P}(X_1 = x_1, X_2 = x_2, X_3 = x_3) = \mathbb{P}(X_1 = x_1)\mathbb{P}(X_2 = x_2 \mid X_1 = x_1)\mathbb{P}(X_3 = x_3 \mid X_1 = x_1, X_2 = x_2).$$

- In words, in order to know the probability of observing a sequence, it suffices to know the probability of observing its first element and the probability of observing each of its following elements given the previous elements.

5 Autoregressive language model

- Recall that a language model assigns a probability to each possible sequence of tokens (text).
- By the chain rule of probability, for every sequence of tokens $x_1, \dots, x_T$, a language model only needs to assign a probability $\mathbb{P}(X_1 = x_1)$ to x_1 being the first token and, for every $t > 1$, a probability $\mathbb{P}(X_t = x_t \mid X_1 = x_1, \dots, X_{t-1} = x_{t-1})$ to x_t being the t-th token if the previous tokens are $x_1, \dots, x_{t-1}$.
- An autoregressive language model can be used to generate text by sampling the first token x_1 from the distribution for X_1 and, for every $t > 1$, sampling the token x_t from the distribution for X_t given $X_1 = x_1, \dots, X_{t-1} = x_{t-1}$.
- Recall that any model that receives its own predictions in order to make additional predictions is considered autoregressive.

6 Markov chain language models

- A first order Markov chain can represent one of the simplest language models.
- For every $T > 1$ and sequence of tokens $x_1, x_2, \dots, x_T$, this model supposes that $\mathbb{P}(X_1 = x_1, X_2 = x_2, \dots, X_T = x_t) > 0$ and, for every $t > 1$,

$$\mathbb{P}(X_t = x_t \mid X_1 = x_1, \dots, X_{t-1} = x_{t-1}) = \mathbb{P}(X_t = x_t \mid X_{t-1} = x_{t-1}).$$

- In words, this model supposes that the current token allows predicting the next token without considering the previous tokens.
- For example, if $T = 3$, a first order Markov chain language model supposes that

$$\mathbb{P}(X_1 = x_1, X_2 = x_2, X_3 = x_3) = \mathbb{P}(X_1 = x_1)\mathbb{P}(X_2 = X_2 \mid X_1 = x_1)\mathbb{P}(X_3 = x_3 \mid X_2 = x_2).$$

- A time-homogeneous first order Markov chain language model supposes that, for every $t > 1$ and pair of tokens x and x',

$$\mathbb{P}(X_t = x' \mid X_{t-1} = x) = p_{x'|x},$$

where $p_{x|x'}$ denotes a parameter associated to the pair of tokens x and x'.

- In words, this model supposes that the current token allows predicting the next token without considering how many tokens came before it.
- For example, a time-homogeneous first order Markov chain language model supposes that

$$\mathbb{P}(X_3 = x' \mid X_2 = x) = p_{x'|x} = \mathbb{P}(X_2 = x' \mid X_1 = x).$$

- There are different approaches to obtain the parameters of a time-homogeneous first order Markov chain language model based on a given text. The following approach is one of the simplest.
- For a given text, let $n_{x,x'}$ denote the number of times that the token x is followed by the token x', and let $n_x = \sum_{x'} n_{x,x'}$ denote the number of times that token x is followed by some token. Furthermore, let $p_{x'|x}$ be given by

$$p_{x'|x} = \frac{n_{x,x'} + \alpha}{n_x + v\alpha},$$

where v is the length of the vocabulary (number of distinct tokens) and $\alpha > 0$ is a small constant that avoids division by zero.

- A time-homogeneous Markov chain language model can be used to generate text by receiving the first token x_1 and, for every $t > 1$, sampling the token x_t from the distribution for X_t given $X_{t-1} = x_{t-1}$.
- An n-gram is a sequence of n tokens. An n-th order Markov chain language model supposes that the current n-gram allows predicting the next token without considering the tokens that preceded it.

- Note that there are v^n possible n-grams, where v is the length of the vocabulary (number of distinct tokens). Therefore, most possible n-grams will not appear in a text if n is relatively large, even if v is small.

[9]:
```python
n = 2

distinct_tokens = len(vocab)
print(f'Distinct tokens in the text: {distinct_tokens}.')

possible_n_grams = distinct_tokens**n
print(f'Possible {n}-grams: {possible_n_grams}.')

existing_n_grams = len(tokens) - n + 1
print(f'Non-distinct {n}-grams in the text: {existing_n_grams}.')

if possible_n_grams > existing_n_grams:
    fraction = 1 - existing_n_grams / possible_n_grams
    print(f'At least {fraction * 100}% of the possible {n}-grams are not in the␣
 ↪text.') # Certainly more if an `n`-gram appears more than once.
```

```
Distinct tokens in the text: 6978.
Possible 2-grams: 48692484.
Non-distinct 2-grams in the text: 75327.
At least 99.84530056014394% of the possible 2-grams are not in the text.
```

6.1 Implementation

- The following code implements an n-th order time-homogeneous Markov chain language model based on the processed text.
- A hyperparameter called temperature is used to control the sampling process. Lowering the temperature causes less probable tokens to be sampled even less frequently, and raising the temperature causes less probable tokens to be sampled more frequently.

[10]:
```python
class MarkovModel:
    def __init__(self, text, order, use_chars, alpha=None):
        self.text = text
        self.order = order
        self.use_chars = use_chars

        self.tokens = tokenize(text, self.use_chars)
        self.vocab = Vocabulary(self.tokens)
        self.indices = np.array(self.vocab[self.tokens])

        self.alpha = alpha
        if self.alpha is None:
            self.alpha = 1. / len(self.vocab)

        self.counts = {} # Stores the number of times that each ngram is␣
↪followed by each token
        for i in range(len(self.indices) - order):
            ngram = tuple(self.indices[i: i + order])
            index = self.indices[i + order]

            self.counts[(ngram, index)] = self.counts.get((ngram, index), 0) + 1

        self.ngram_token_freqs = sorted(self.counts.items(), key=lambda x:␣
↪x[1], reverse=True)

    def pseudocounts(self, ngram):
        return np.array([self.counts.get((ngram, index), 0) + self.alpha for␣
↪index in range(len(self.vocab))]) # An array with the number of times that␣
↪`ngram` was followed by each token, plus `self.alpha`

    def generate(self, start_text, n_tokens, temperature):
        tokens = tokenize(start_text, self.use_chars)

        if len(tokens) < self.order:
            raise Exception(f'You must provide at least {self.order} tokens to␣
↪generate text')

        indices = self.vocab[tokens]
        for _ in range(n_tokens):
            ngram = tuple(indices[-self.order:])

            pseudocounts = self.pseudocounts(ngram)

            # The following two lines change the probability of sampling tokens␣
↪based on the `temperature`
            pseudocounts = pseudocounts / np.max(pseudocounts)
            pseudocounts = np.exp(pseudocounts / temperature)

            p = pseudocounts / np.sum(pseudocounts)

            token = np.random.choice(len(self.vocab), p=p)

            indices.append(token)

        tokens = self.vocab.to_tokens(indices)

        return ''.join(tokens) if self.use_chars else ' '.join(tokens)

mm = MarkovModel(text=text, order=4, use_chars=False) # Choose the order and␣
↪whether to assign one token for each character or one token for each␣
↪sequence of letters
```

```
print(f'Most frequent pairs of {mm.order}-gram and token:')
for (ngram, token), freq in mm.ngram_token_freqs[:10]:
    print(f'{tuple(mm.vocab.to_tokens(ngram))} -> {mm.vocab.to_tokens(token)}␣
  ↪({freq}).')

print('\nGenerated text: ')
mm.generate('i cannot describe to', n_tokens=150, temperature=0.01)
```

```
Most frequent pairs of 4-gram and token:
('chapter', 'chapter', 'chapter', 'chapter') -> chapter (20).
('letter', 'to', 'mrs', 'saville') -> england (4).
('be', 'with', 'you', 'on') -> your (4).
('with', 'you', 'on', 'your') -> wedding (4).
('you', 'on', 'your', 'wedding') -> night (4).
('i', 'cannot', 'describe', 'to') -> you (3).
('for', 'my', 'own', 'part') -> i (3).
('for', 'a', 'long', 'time') -> i (3).
('was', 'to', 'convey', 'me') -> away (3).
('monster', 'whom', 'i', 'had') -> created (3).

Generated text:
```

```
[10]: 'i cannot describe to you the agony that these reflections inflicted upon me i
      tried to dispel them but sorrow only increased with knowledge oh that i had for
      ever remained in my native wood nor known nor felt beyond the sensations of
      hunger thirst and heat of what a strange nature is knowledge it clings to the
      mind when it has once seized on it like a lichen on the rock i wished sometimes
      to shake off all thought and feeling but i learned that there was but one means
      to overcome the sensation of pain and that was death a state which i feared yet
      did not understand i admired virtue and good feelings and loved the gentle
      manners and amiable qualities of my cottagers but i was shut out from
      intercourse with them except through means which i obtained by stealth when i
      was unseen and unknown and which rather increased'
```

7 Neural language models

- Consider the following two texts:
  1. “The banana is ripe.”
  2. “The strawberry is ripe.”
- A good language model trained on a large corpora should be able infer that interchanging “bananas” for “strawberries” often results in a meaningful sentence, since both words refer to edible fruits.
- Therefore, a good language model could assign a high probability to the second text even if it was not part of the large training corpora.

- Markov chain language models do not enable such generalization.
- More powerful language models, such as language models based on neural networks, may be able to address this issue.

8 Classification for next token prediction

- Let $T > 1$ and consider a sequence $\mathbf{s}_T = \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T$, where each $\mathbf{x}_i \in \mathbb{R}^v$ is a vector that corresponds to one-hot encoding a token x_i from a vocabulary with v tokens.
- Let $\mathbf{s}_t = \mathbf{x}_1, \dots, \mathbf{x}_t$ denote a sequence that contains the first $t < T$ elements of the sequence $\mathbf{s}_T$.
- If each $\mathbf{s}_t$ is interpreted as an observation, the task of predicting x_{t+1} given $\mathbf{s}_t$ is a classification task.
- In the next lectures, we will see how a neural network can make a prediction $\hat{\mathbf{x}}_{t+1}$ based on an entire sequence of vectors $\mathbf{s}_t = \mathbf{x}_1, \dots, \mathbf{x}_t$.

which corresponds to averaging the cross entropy loss for each of the $T - 1$ predictions.

9 Partitioning and batching

- If the purpose of a language model is to generate entire books, each sequence $\mathbf{s}_T$ in a training dataset may correspond to an entire book.
- If the purpose of a language model is to generate passages from books, each sequence $\mathbf{s}_T$ in the training dataset may correspond to a passage from a book.
- Therefore, there are different ways to organize a corpora into a training dataset for the purpose of training a language model.
- In what follows, we will assume that every text in a given corpora has been concatenated into a single text, which was then processed into a single sequence of token indices $a = a_1, a_2, \dots, a_L$.
- We will cover two methods for organizing the sequence a into a training dataset: **random partitioning** and **sequential partitioning**.
- For a pre-defined batch size B and a pre-defined number of steps T, the sequence a will be partitioned into subsequences of length T, which will be grouped into batches of size B.
- More concretely, each epoch of these methods will produce a sequence of pairs $(\mathbf{X}_1, \mathbf{Y}_1), (\mathbf{X}_2, \mathbf{Y}_2), \dots, (\mathbf{X}_n, \mathbf{Y}_n)$, where $\mathbf{X}_j$ and $\mathbf{Y}_j$ are $B \times T$ matrices.
- The i-th row of the matrix $\mathbf{X}_j$ will contain a subsequence $a_{k+1}, a_{k+2}, \dots, a_{k+T}$ with T elements from the sequence a, where k is an index that depends on i, j, and the partitioning method. In this case, the i-th row of the matrix $\mathbf{Y}_j$ would contain the subsequence $a_{k+2}, a_{k+3}, \dots, a_{k+T+1}$.
- In words, for the purpose of training a language model, the first t elements of each row of the matrix $\mathbf{X}_j$ will be used to predict the t-th element of the corresponding row of the matrix $\mathbf{Y}_j$.
- There are two reasons for partitioning and batching the original sequence a in this way:

  – Training the neural networks that we will study in the following lectures using long sequences will be expensive in comparison with computing predictions using long sequences.
  – For the same reasons that previous implementations made predictions for batches of vectors (rather than single vectors), the vectorized implementations of these neural networks make predictions for entire batches of sequences (rather than single sequences).

9.1 Random partitioning

- The following code implements random partitioning using a Python generator.

[11]:
```
def random_partitioning(sequence, batch_size, num_steps, offset=None):
    if offset is None:
        offset = np.random.randint(num_steps) # A random number between 0 and␣
 ↪`num_steps` (excluding `num_steps`)

    sequence = sequence[offset:] # Discards the first `offset` elements of the␣
 ↪sequence

    num_subseqs = (len(sequence) - 1) // num_steps # The number of subsequences␣
 ↪of length `num_steps` that fit into the `sequence`

    subseq_indices = list(range(0, num_subseqs * num_steps, num_steps)) # A␣
 ↪list with the index where each subsequence starts: [0, `num_steps`, 2 *␣
 ↪`num_steps`, ...]

    np.random.shuffle(subseq_indices) # Randomizes the indices to enable␣
 ↪creating batches composed of randomly chosen subsequences

    num_batches = num_subseqs // batch_size # The number of batches required to␣
 ↪fit every subsequences

    for i in range(0, num_batches * batch_size, batch_size):
        batch_indices = subseq_indices[i: i + batch_size] # The index where␣
 ↪each subsequence that should be part of this batch starts

        X = [sequence[j: j + num_steps] for j in batch_indices] # Organizes␣
 ↪each subsequence into a row
        Y = [sequence[j + 1: j + 1 + num_steps] for j in batch_indices] #␣
 ↪Organizes each subsequence (shifted by one index) into a row

        yield torch.tensor(X), torch.tensor(Y) # Yields a pair of matrices


sequence = list(range(31))
print(f'Partitioning and batching the sequence {sequence}:')
```

```
for i, (X, Y) in enumerate(random_partitioning(sequence, batch_size=3,␣
↪num_steps=5, offset=0)):
    print(f'X_{i + 1}: \n{X}.')
    print(f'Y_{i + 1}: \n{Y}.\n')
    print()


print('Partitioning and batching the sequence of token indices (first two␣
↪batches, converted into batches of tokens):')
for i, (X, Y) in enumerate(random_partitioning(indices, batch_size=3,␣
↪num_steps=5)):
    X_tokens = np.array([vocab.to_tokens(list(X[j])) for j in range(X.
↪shape[0])])
    Y_tokens = np.array([vocab.to_tokens(list(Y[j])) for j in range(Y.
↪shape[0])])
    print(f'X_{1}: \n{X_tokens}.')
    print(f'Y_{1}: \n{Y_tokens}.\n')
    print()

    if i >= 1:
        break
```

```
Partitioning and batching the sequence [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]:
X_1:
tensor([[20, 21, 22, 23, 24],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14]]).
Y_1:
tensor([[21, 22, 23, 24, 25],
        [ 6,  7,  8,  9, 10],
        [11, 12, 13, 14, 15]]).



X_2:
tensor([[25, 26, 27, 28, 29],
        [ 0,  1,  2,  3,  4],
        [15, 16, 17, 18, 19]]).
Y_2:
tensor([[26, 27, 28, 29, 30],
        [ 1,  2,  3,  4,  5],
        [16, 17, 18, 19, 20]]).



Partitioning and batching the sequence of token indices (first two batches,
converted into batches of tokens):
X_1:
```

```
[['i' 'could' 'banish' 'disease' 'from']
 ['a' 'blasted' 'tree' 'the' 'bolt']
 ['the' 'fire' 'the' 'wet' 'wood']].
Y_1:
[['could' 'banish' 'disease' 'from' 'the']
 ['blasted' 'tree' 'the' 'bolt' 'has']
 ['fire' 'the' 'wet' 'wood' 'which']].


X_1:
[['and' 'rage' 'choked' 'my' 'utterance']
 ['hiding' 'places' 'who' 'shall' 'conceive']
 ['die' 'and' 'at' 'once' 'satisfy']].
Y_1:
[['rage' 'choked' 'my' 'utterance' 'i']
 ['places' 'who' 'shall' 'conceive' 'the']
 ['and' 'at' 'once' 'satisfy' 'and']].
```

9.2 Sequential Partitioning

- The following code implements sequential partitioning using a Python generator.
- Most notably, if the i-th row of the matrix X_j contains a subsequence $a_{k+1}, a_{k+2}, \dots, a_{k+T}$ with T elements from the sequence a, the i-th row of the matrix X_{j+1} will contain the adjacent subsequence $a_{k+1+T}, a_{k+2+T}, \dots, a_{k+2T}$.

[12]:
```python
def sequential_partitioning(sequence, batch_size, num_steps, offset=None):
    if offset is None:
        offset = np.random.randint(num_steps) # A random number between 0 and␣
 ↪`num_steps` (excluding `num_steps`)

    sequence = sequence[offset:] # Discards the first `offset` elements of the␣
 ↪sequence

    len_macro_subseq = (len(sequence) - 1) // batch_size # The length of each␣
 ↪macro-subsequence. There is one macro-subsequence for each unit of batch size

    num_elements = len_macro_subseq * batch_size # The number of elements that␣
 ↪fit into the macro-subsequences

    Xs = torch.tensor(sequence[: num_elements]) # Selects the elements that fit␣
 ↪into the macro-subsequences
    Ys = torch.tensor(sequence[1: num_elements + 1]) # Select the elements that␣
 ↪fit into the macro-subsequences (shifted by one index)
```

```
    Xs = Xs.reshape(batch_size, -1) # Each row of this matrix corresponds to a␣
 ↪macro-subsequence
    Ys = Ys.reshape(batch_size, -1) # Each row of this matrix corresponds to a␣
 ↪macro-subsequence (shifted by one index)

    num_subseqs = Xs.shape[1] // num_steps # The number of subsequences that␣
 ↪fit into each macro-subsequence

    for i in range(0, num_subseqs * num_steps, num_steps):
        X = Xs[:, i: i + num_steps] # Each row of `X` contains a subsequence␣
 ↪from a distinct macro-subsequence
        Y = Ys[:, i: i + num_steps] # Each row of `Y` contains a subsequence␣
 ↪from a distinct macro-subsequence (shifted by one index)

        yield X, Y # Yields a pair of matrices


sequence = list(range(31))
print(f'Partitioning and batching the sequence {sequence}:')
for i, (X, Y) in enumerate(sequential_partitioning(sequence, batch_size=3,␣
 ↪num_steps=5, offset=0)):
    print(f'X_{i + 1}: \n{X}.')
    print(f'Y_{i + 1}: \n{Y}.\n')
    print()


print('Partitioning and batching the sequence of token indices (first two␣
 ↪batches, converted into batches of tokens):')
for i, (X, Y) in enumerate(sequential_partitioning(indices, batch_size=3,␣
 ↪num_steps=5)):
    X_tokens = np.array([vocab.to_tokens(list(X[j])) for j in range(X.
 ↪shape[0])])
    Y_tokens = np.array([vocab.to_tokens(list(Y[j])) for j in range(Y.
 ↪shape[0])])
    print(f'X_{i + 1}: \n{X_tokens}.')
    print(f'Y_{i + 1}: \n{Y_tokens}.\n')
    print()

    if i >= 1:
        break
```

```
Partitioning and batching the sequence [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]:
X_1:
tensor([[ 0,  1,  2,  3,  4],
        [10, 11, 12, 13, 14],
        [20, 21, 22, 23, 24]]).
```

```
Y_1:
tensor([[ 1,  2,  3,  4,  5],
        [11, 12, 13, 14, 15],
        [21, 22, 23, 24, 25]]).


X_2:
tensor([[ 5,  6,  7,  8,  9],
        [15, 16, 17, 18, 19],
        [25, 26, 27, 28, 29]]).
Y_2:
tensor([[ 6,  7,  8,  9, 10],
        [16, 17, 18, 19, 20],
        [26, 27, 28, 29, 30]]).


Partitioning and batching the sequence of token indices (first two batches,
converted into batches of tokens):
X_1:
[['the' 'modern' 'prometheus' 'by' 'mary']
 ['i' 'had' 'before' 'experienced' 'sensations']
 ['arrived' 'at' 'mainz' 'the' 'course']].
Y_1:
[['modern' 'prometheus' 'by' 'mary' 'wollstonecraft']
 ['had' 'before' 'experienced' 'sensations' 'of']
 ['at' 'mainz' 'the' 'course' 'of']].


X_2:
[['wollstonecraft' 'godwin' 'shelley' 'contents' 'letter']
 ['of' 'horror' 'and' 'i' 'have']
 ['of' 'the' 'rhine' 'below' 'mainz']].
Y_2:
[['godwin' 'shelley' 'contents' 'letter' 'letter']
 ['horror' 'and' 'i' 'have' 'endeavoured']
 ['the' 'rhine' 'below' 'mainz' 'becomes']].
```

10 Recommended reading

- Dive Into Deep Learning: Chapters 9.1, 9.2, and 9.3.

11 [Storing this notebook as a pdf]

[16]:
```
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)

!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic

# Set the path to this notebook below (add \ before spaces). The output `pdf`␣
↪will be stored in the corresponding folder.
!jupyter nbconvert --to pdf /content/gdrive/My\ Drive/Colab\ Notebooks/nndl/
↪week_09/lecture/01_Sequence_Processing_and_Language_Models.ipynb

# If having issues, save this notebook (File > Save) and restart the session␣
↪(Runtime > Restart session) before running this cell. To debug, remove the␣
↪first line (`%%capture`).
```

```
Mounted at /content/gdrive
Reading package lists… Done
Building dependency tree… Done
Reading state information… Done
texlive-fonts-recommended is already the newest version (2021.20220204-1).
texlive-plain-generic is already the newest version (2021.20220204-1).
texlive-xetex is already the newest version (2021.20220204-1).
0 upgraded, 0 newly installed, 0 to remove and 33 not upgraded.
[NbConvertApp] Converting notebook /content/gdrive/My Drive/Colab
Notebooks/nndl/week_09/lecture/01_Sequence_Processing_and_Language_Models.ipynb
to pdf
[NbConvertApp] Writing 108151 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', 'notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', 'notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 115403 bytes to /content/gdrive/My Drive/Colab
Notebooks/nndl/week_09/lecture/01_Sequence_Processing_and_Language_Models.pdf
```