Recurrent Neural Networks

- So far we encountered two types of data: tabular data and image data.
- However, there are other types of data following a sequential order:
  - words in sentences
  - image frames in a video
  - the audio signal in a conversation
  - the browsing behavior on a website
- It is reasonable to assume that specialized models for such data will do better at describing them.
- In short, while CNNs can efficiently process spatial information, *recurrent neural networks* (RNNs) are designed to better handle sequential information.

Text Preprocessing

- Text is one of the most popular examples of sequence data.
- For example, an article can be simply viewed as a sequence of words, or even a sequence of characters.
- Common preprocessing steps for text include:
    1. Load text as strings into memory.
    2. Split strings into tokens (e.g., words and characters).
    3. Build a table of vocabulary to map the split tokens to numerical indices.
    4. Convert text into sequences of numerical indices so they can be manipulated by models easily.

Reading the Dataset

- We will use the text from H. G. Wells' *The Time Machine*.
- A fairly small corpus of just over 30000 words
    - More realistic document collections contain many billions of words.

Put everything together

In [49]:
```
batch_size, num_steps = 32, 35
train_iter, vocab = mu.load_data_time_machine(batch_size, num_steps)
```

In [50]:
```
train_iterator = iter(train_iter)
batch_1 = next(train_iterator)
```

In [51]:
```
sample_1 = batch_1[0][0,:]
print(sample_1)
```

```
tensor([ 2,  1, 21, 19,  1,  9,  1, 18,  1, 17,  2, 12, 12,  8,  1,  5,  3,
9,
         2,  1,  3,  5, 13,  2,  1,  3, 10,  4, 22,  2, 12, 12,  2, 10,  1])
```

In [52]:
```
labels_1 = batch_1[1][0,:]
print(labels_1)
```

```
tensor([ 1, 21, 19,  1,  9,  1, 18,  1, 17,  2, 12, 12,  8,  1,  5,  3,  9,
2,
         1,  3,  5, 13,  2,  1,  3, 10,  4, 22,  2, 12, 12,  2, 10,  1, 16])
```

In [3]:

```python
# The following function reads the dataset into a list of text lines, where each line is a string.
# For simplicity, punctuation and capitalization are ignored
mu.DATA_HUB['time_machine'] = (mu.DATA_URL + 'timemachine.txt',
                                '090b5e7e70c295757f55df93cb0a180b9691891a')

def read_time_machine():  #@save
    """Load the time machine dataset into a list of text lines."""
    with open(mu.download('time_machine'), 'r') as f:
        lines = f.readlines()
    return [re.sub('[^A-Za-z]+', ' ', line).strip().lower() for line in lines]

lines = read_time_machine()
print(f'# text lines: {len(lines)}')
print(lines[0])
print(lines[10])
```

```
# text lines: 3221
the time machine by h g wells
twinkled and his usually pale face was flushed and animated the
```

Tokenization

- The following `tokenize` function takes a list (`lines`) as the input, where each list is a text sequence (e.g., a text line).
- Each text sequence is split into a list of tokens.
- A *token* is the basic unit in text.
- In the end, a list of token lists are returned, where each token is a string.

In [21]:
```python
def tokenize(lines, token='word'):  #@save
    """Split text lines into word or character tokens."""
    if token == 'word':
        return [line.split() for line in lines]
    elif token == 'char':
        return [list(line) for line in lines]
    else:
        print('ERROR: unknown token type: ' + token)

tokens = tokenize(lines)
for i in range(11):
    print(tokens[i])
```

```
['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
[]
[]
[]
[]
['i']
[]
[]
['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be', 'convenient',
'to', 'speak', 'of', 'him']
['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his', 'grey',
'eyes', 'shone', 'and']
['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and
', 'animated', 'the']
```

Vocabulary

- The string type of the token is inconvenient to be used by models, which take numerical inputs.
- A *vocabulary* is a dictionary for mapping string tokens into numerical indices starting from 0.
    - First count the unique tokens in all the documents from the training set, also called a *corpus*,
        - Then assign a numerical index to each unique token.
        - Rarely appeared tokens are often removed to reduce the complexity.
        - Any token that does not exist in the corpus or has been removed is mapped into a special unknown token “<unk>”.
        - Optional: add a list of reserved tokens, such as “<pad>” for padding, “<bos>” to present the beginning for a sequence, and “<eos>” for the end of a sequence.

In [9]:
```
class Vocab:  #@save
    """Vocabulary for text."""
    def __init__(self, tokens=None, min_freq=0, reserved_tokens=None):
        if tokens is None:
            tokens = []
        if reserved_tokens is None:
            reserved_tokens = []
        # Sort according to frequencies
        counter = count_corpus(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[0])
        self.token_freqs.sort(key=lambda x: x[1], reverse=True)
        # The index for the unknown token is 0
        self.unk, uniq_tokens = 0, ['<unk>'] + reserved_tokens
        uniq_tokens += [token for token, freq in self.token_freqs
                        if freq >= min_freq and token not in uniq_tokens]
        self.idx_to_token, self.token_to_idx = [], dict()
        for token in uniq_tokens:
            self.idx_to_token.append(token)
            self.token_to_idx[token] = len(self.idx_to_token) - 1

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

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):
        if not isinstance(indices, (list, tuple)):
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]
```

In [22]:
```
# Construct a vocabulary using the time machine dataset as the corpus.
vocab = Vocab(tokens)
# Print the first few frequent tokens with their indices.
print(list(vocab.token_to_idx.items())[:10])
```

```
[('<unk>', 0), ('the', 1), ('i', 2), ('and', 3), ('of', 4), ('a', 5), ('to',
6), ('was', 7), ('in', 8), ('that', 9)]
```

In [23]:
```
# Convert each text line into a list of numerical indices.
for i in [0, 10]:
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])
```

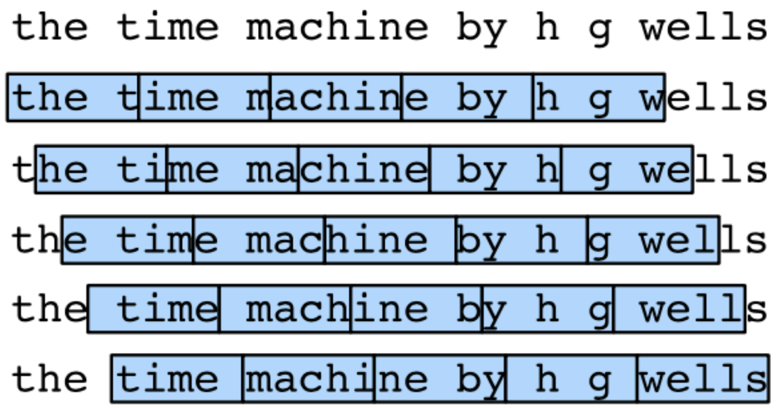
```
words: ['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
indices: [1, 19, 50, 40, 3130, 3058, 438]
words: ['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed
', 'and', 'animated', 'the']
indices: [4399, 3, 25, 1398, 387, 113, 7, 1676, 3, 1053, 1]
```

Putting Everything Together

- Using the above functions, we package everything into the `load_corpus_time_machine` function, which returns `corpus`, a list of token indices, and `vocab`, the vocabulary of the time machine corpus.
- The modifications we did here are:
    1. we tokenize text into characters, not words, to simplify the training in later sections;
    2. `corpus` is a single list, not a list of token lists, since each text line in the time machine dataset is not necessarily a sentence or a paragraph.

Reading Long Sequence Data

- Since a text sequence can be arbitrarily long, we will partition it into subsequences with the same number of time steps.
- When training our neural network, a minibatch of such subsequences will be fed into the model.
- Suppose that the network processes a subsequence of n time steps at a time.
- The figure below shows all the different ways to obtain subsequences from an original text sequence, where $n = 5$ and a token at each time step corresponds to a character.
- We could pick any arbitrary offset that indicates the initial position.
- In practice we pick a random offset to partition a sequence



Random Sampling

- Each example is a subsequence from the original long sequence.
- The subsequences from two adjacent random minibatches are not necessarily adjacent in the original sequence.
- For language modeling, the target is to predict the next token, hence the labels are the original sequence, shifted by one token.

In [33]:
```python
# `num_steps` is the predefined number of time steps in each subsequence
def seq_data_iter_random(corpus, batch_size, num_steps):  #@save
    """Generate a minibatch of subsequences using random sampling."""
    # Start with a random offset to partition a sequence
    corpus = corpus[random.randint(0, num_steps):]
    # Subtract 1 since we need to account for labels
    num_subseqs = (len(corpus) - 1) // num_steps
    # The starting indices for subsequences of length `num_steps`
    initial_indices = list(range(0, num_subseqs * num_steps, num_steps))
    # In random sampling, the subsequences from two adjacent random
    # minibatches during iteration are not necessarily adjacent on the
    # original sequence
    random.shuffle(initial_indices)

    def data(pos):
        # Return a sequence of length `num_steps` starting from `pos`
        return corpus[pos: pos + num_steps]

    num_subseqs_per_example = num_subseqs // batch_size
    for i in range(0, batch_size * num_subseqs_per_example, batch_size):
        # Here, `initial_indices` contains randomized starting indices for
        # subsequences
        initial_indices_per_batch = initial_indices[i: i + batch_size]
        X = [data(j) for j in initial_indices_per_batch]
        Y = [data(j + 1) for j in initial_indices_per_batch]
        yield torch.tensor(X), torch.tensor(Y)
```

- Example: Generate a sequence from 0 to 34. Batch size and numbers of time steps are 2 and 5,
- This means that we can generate $\lfloor (35-1)/5 \rfloor = 6$ feature-label subsequence pairs.
- With a minibatch size of 2, we only get 3 minibatches.

In [34]:
```
my_seq = list(range(35))
for X, Y in seq_data_iter_random(my_seq, batch_size=2, num_steps=5):
    print('X: ', X, '\nY:', Y)
```

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

Sequential Partitioning

- Ensures that the subsequences from two adjacent minibatches during iteration are adjacent in the original sequence.
- This strategy preserves the order of split subsequences when iterating over minibatches

In [35]:
```python
def seq_data_iter_sequential(corpus, batch_size, num_steps):  #@save
    """Generate a minibatch of subsequences using sequential partitioning."""
    # Start with a random offset to partition a sequence
    offset = random.randint(0, num_steps)
    num_tokens = ((len(corpus) - offset - 1) // batch_size) * batch_size
    Xs = torch.tensor(corpus[offset: offset + num_tokens])
    Ys = torch.tensor(corpus[offset + 1: offset + 1 + num_tokens])
    Xs, Ys = Xs.reshape(batch_size, -1), Ys.reshape(batch_size, -1)
    num_batches = Xs.shape[1] // num_steps
    for i in range(0, num_batches * num_steps, num_steps):
        X = Xs[:, i: i + num_steps]
        Y = Ys[:, i: i + num_steps]
        yield X, Y
```

```
In [36]: # Previous example using sequential sampling
         for X, Y in seq_data_iter_sequential(my_seq, batch_size=2, num_steps=5):
             print('X: ', X, '\nY:', Y)
```

```
X:  tensor([[ 0,  1,  2,  3,  4],
        [17, 18, 19, 20, 21]])
Y: tensor([[ 1,  2,  3,  4,  5],
        [18, 19, 20, 21, 22]])
X:  tensor([[ 5,  6,  7,  8,  9],
        [22, 23, 24, 25, 26]])
Y: tensor([[ 6,  7,  8,  9, 10],
        [23, 24, 25, 26, 27]])
X:  tensor([[10, 11, 12, 13, 14],
        [27, 28, 29, 30, 31]])
Y: tensor([[11, 12, 13, 14, 15],
        [28, 29, 30, 31, 32]])
```

Loading the data

- We use the above functions to define our sequence dataloader

In [37]:
```
class SeqDataLoader:
    """An iterator to load sequence data."""
    def __init__(self, batch_size, num_steps, use_random_iter, max_tokens):
        if use_random_iter:
            self.data_iter_fn = mu.seq_data_iter_random
        else:
            self.data_iter_fn = mu.seq_data_iter_sequential
        self.corpus, self.vocab = mu.load_corpus_time_machine(max_tokens)
        self.batch_size, self.num_steps = batch_size, num_steps

    def __iter__(self):
        return self.data_iter_fn(self.corpus, self.batch_size, self.num_steps)
```

In [38]:
```
# similar to load_data_fashion_mnist
def load_data_time_machine(batch_size, num_steps,
                           use_random_iter=False, max_tokens=10000):
    """Return the iterator and the vocabulary of the time machine dataset."""
    data_iter = SeqDataLoader(
        batch_size, num_steps, use_random_iter, max_tokens)
    return data_iter, data_iter.vocab
```

Put everything together

In [49]:
```
batch_size, num_steps = 32, 35
train_iter, vocab = mu.load_data_time_machine(batch_size, num_steps)
```

In [50]:
```
train_iterator = iter(train_iter)
batch_1 = next(train_iterator)
```

In [51]:
```
sample_1 = batch_1[0][0,:]
print(sample_1)
```

```
tensor([ 2,  1, 21, 19,  1,  9,  1, 18,  1, 17,  2, 12, 12,  8,  1,  5,  3,
9,
         2,  1,  3,  5, 13,  2,  1,  3, 10,  4, 22,  2, 12, 12,  2, 10,  1])
```

In [52]:
```
labels_1 = batch_1[1][0,:]
print(labels_1)
```

```
tensor([ 1, 21, 19,  1,  9,  1, 18,  1, 17,  2, 12, 12,  8,  1,  5,  3,  9,
2,
         1,  3,  5, 13,  2,  1,  3, 10,  4, 22,  2, 12, 12,  2, 10,  1, 16])
```