# 8 Recurrent Neural Networks

So far we encountered two types of data: tabular data and image data. For the latter we designed specialized layers to take advantage of the regularity in them. In other words, if we were to permute the pixels in an image, it would be much more difficult to reason about its content of something that would look much like the background of a test pattern in the times of analog TV.

Most importantly, so far we tacitly assumed that our data are all drawn from some distribution, and all the examples are independently and identically distributed (i.i.d.). Unfortunately, this is not true for most data. For instance, the words in this paragraph are written in sequence, and it would be quite difficult to decipher its meaning if they were permuted randomly. Likewise, image frames in a video, the audio signal in a conversation, and the browsing behavior on a website, all follow sequential order. It is thus reasonable to assume that specialized models for such data will do better at describing them.

Another issue arises from the fact that we might not only receive a sequence as an input but rather might be expected to continue the sequence. For instance, the task could be to continue the series $2, 4, 6, 8, 10, \ldots$ This is quite common in time series analysis, to predict the stock market, the fever curve of a patient, or the acceleration needed for a race car. Again we want to have models that can handle such data.

In short, while CNNs can efficiently process spatial information, *recurrent neural networks* (RNNs) are designed to better handle sequential information. RNNs introduce state variables to store past information, together with the current inputs, to determine the current outputs.

Many of the examples for using recurrent networks are based on text data. Hence, we will emphasize language models in this chapter. After a more formal review of sequence data we introduce practical techniques for preprocessing text data. Next, we discuss basic concepts of a language model and use this discussion as the inspiration for the design of RNNs. In the end, we describe the gradient calculation method for RNNs to explore problems that may be encountered when training such networks.

## 8.1 Sequence Models

Imagine that you are watching movies on Netflix. As a good Netflix user, you decide to rate each of the movies religiously. After all, a good movie is a good movie, and you want to watch more of them, right? As it turns out, things are not quite so simple. People's opinions on movies can change quite significantly over time. In fact, psychologists even have names for some of the effects:

- There is *anchoring*, based on someone else's opinion. For instance, after the Oscar awards, ratings for the corresponding movie go up, even though it is still the same movie. This effect persists for a few months until the award is forgotten. It has been shown that the effect lifts rating by over half a point (Wu et al., 2017).

- There is the *hedonic adaptation*, where humans quickly adapt to accept an improved or a worsened situation as the new normal. For instance, after watching many good movies, the expectations that the next movie is equally good or better are high. Hence, even an average movie might be considered as bad after many great ones are watched.
- There is *seasonality*. Very few viewers like to watch a Santa Claus movie in August.
- In some cases, movies become unpopular due to the misbehaviors of directors or actors in the production.
- Some movies become cult movies, because they were almost comically bad. *Plan 9 from Outer Space* and *Troll 2* achieved a high degree of notoriety for this reason.

In short, movie ratings are anything but stationary. Thus, using temporal dynamics led to more accurate movie recommendations (Koren, 2009). Of course, sequence data are not just about movie ratings. The following gives more illustrations.

- Many users have highly particular behavior when it comes to the time when they open apps. For instance, social media apps are much more popular after school with students. Stock market trading apps are more commonly used when the markets are open.
- It is much harder to predict tomorrow's stock prices than to fill in the blanks for a stock price we missed yesterday, even though both are just a matter of estimating one number. After all, foresight is so much harder than hindsight. In statistics, the former (predicting beyond the known observations) is called *extrapolation* whereas the latter (estimating between the existing observations) is called *interpolation*.
- Music, speech, text, and videos are all sequential in nature. If we were to permute them they would make little sense. The headline *dog bites man* is much less surprising than *man bites dog*, even though the words are identical.
- Earthquakes are strongly correlated, i.e., after a massive earthquake there are very likely several smaller aftershocks, much more so than without the strong quake. In fact, earthquakes are spatiotemporally correlated, i.e., the aftershocks typically occur within a short time span and in close proximity.
- Humans interact with each other in a sequential nature, as can be seen in Twitter fights, dance patterns, and debates.

### 8.1.1 Statistical Tools

We need statistical tools and new deep neural network architectures to deal with sequence data. To keep things simple, we use the stock price (FTSE 100 index) illustrated in Fig. 8.1.1 as an example.

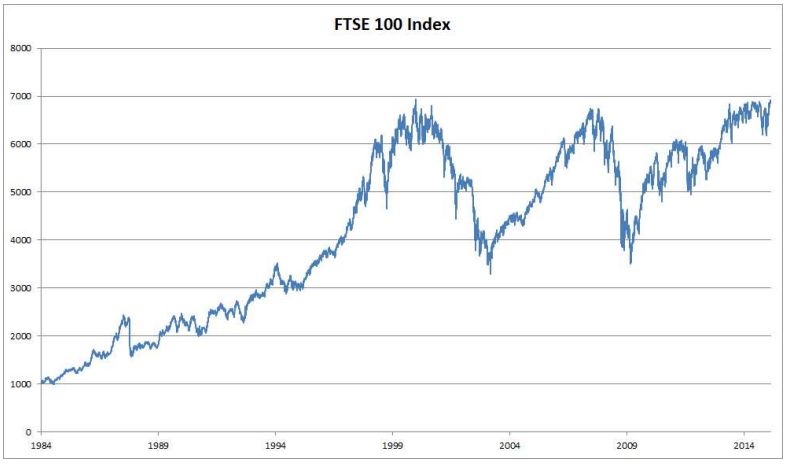



Fig. 8.1.1: FTSE 100 index over about 30 years.

Let us denote the prices by $x_t$, i.e., at *time step* $t \in \mathbb{Z}^+$ we observe price $x_t$. Note that for sequences in this text, $t$ will typically be discrete and vary over integers or its subset. Suppose that a trader who wants to do well in the stock market on day $t$ predicts $x_t$ via

$$x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1). \tag{8.1.1}$$

#### Autoregressive Models

In order to achieve this, our trader could use a regression model such as the one that we trained in Section 3.3. There is just one major problem: the number of inputs, $x_{t-1}, \ldots, x_1$ varies, depending on $t$. That is to say, the number increases with the amount of data that we encounter, and we will need an approximation to make this computationally tractable. Much of what follows in this chapter will revolve around how to estimate $P(x_t \mid x_{t-1}, \ldots, x_1)$ efficiently. In a nutshell it boils down to two strategies as follows.

First, assume that the potentially rather long sequence $x_{t-1}, \ldots, x_1$ is not really necessary. In this case we might content ourselves with some timespan of length $\tau$ and only use $x_{t-1}, \ldots, x_{t-\tau}$ observations. The immediate benefit is that now the number of arguments is always the same, at least for $t > \tau$. This allows us to train a deep network as indicated above. Such models will be called *autoregressive models*, as they quite literally perform regression on themselves.

The second strategy, shown in Fig. 8.1.2, is to keep some summary $h_t$ of the past observations, and at the same time update $h_t$ in addition to the prediction $\hat{x}_t$. This leads to models that estimate $x_t$ with $\hat{x}_t = P(x_t \mid h_t)$ and moreover updates of the form $h_t = g(h_{t-1}, x_{t-1})$. Since $h_t$ is never observed, these models are also called *latent autoregressive models*.

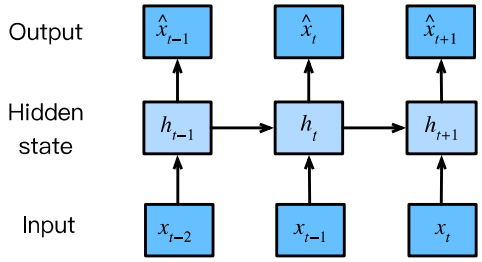



Fig. 8.1.2: A latent autoregressive model.

Both cases raise the obvious question of how to generate training data. One typically uses historical observations to predict the next observation given the ones up to right now. Obviously we do not expect time to stand still. However, a common assumption is that while the specific values of $x_t$ might change, at least the dynamics of the sequence itself will not. This is reasonable, since novel dynamics are just that, novel and thus not predictable using data that we have so far. Statisticians call dynamics that do not change *stationary*. Regardless of what we do, we will thus get an estimate of the entire sequence via

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}, \ldots, x_1). \tag{8.1.2}$$

Note that the above considerations still hold if we deal with discrete objects, such as words, rather than continuous numbers. The only difference is that in such a situation we need to use a classifier rather than a regression model to estimate $P(x_t \mid x_{t-1}, \ldots, x_1)$.

### Markov Models

Recall the approximation that in an autoregressive model we use only $x_{t-1}, \ldots, x_{t-\tau}$ instead of $x_{t-1}, \ldots, x_1$ to estimate $x_t$. Whenever this approximation is accurate we say that the sequence satisfies a *Markov condition*. In particular, if $\tau = 1$, we have a *first-order Markov model* and $P(x)$ is given by

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}) \text{ where } P(x_1 \mid x_0) = P(x_1). \tag{8.1.3}$$

Such models are particularly nice whenever $x_t$ assumes only a discrete value, since in this case dynamic programming can be used to compute values along the chain exactly. For instance, we can compute $P(x_{t+1} \mid x_{t-1})$ efficiently:

$$\begin{aligned} P(x_{t+1} \mid x_{t-1}) &= \frac{\sum_{x_t} P(x_{t+1}, x_t, x_{t-1})}{P(x_{t-1})} \\ &= \frac{\sum_{x_t} P(x_{t+1} \mid x_t, x_{t-1}) P(x_t, x_{t-1})}{P(x_{t-1})} \\ &= \sum_{x_t} P(x_{t+1} \mid x_t) P(x_t \mid x_{t-1}) \end{aligned} \tag{8.1.4}$$

by using the fact that we only need to take into account a very short history of past observations: $P(x_{t+1} \mid x_t, x_{t-1}) = P(x_{t+1} \mid x_t)$. Going into details of dynamic programming is beyond the scope of this section. Control and reinforcement learning algorithms use such tools extensively.

### Causality

In principle, there is nothing wrong with unfolding $P(x_1, \ldots, x_T)$ in reverse order. After all, by conditioning we can always write it via

$$P(x_1, \ldots, x_T) = \prod_{t=T}^{1} P(x_t \mid x_{t+1}, \ldots, x_T). \tag{8.1.5}$$

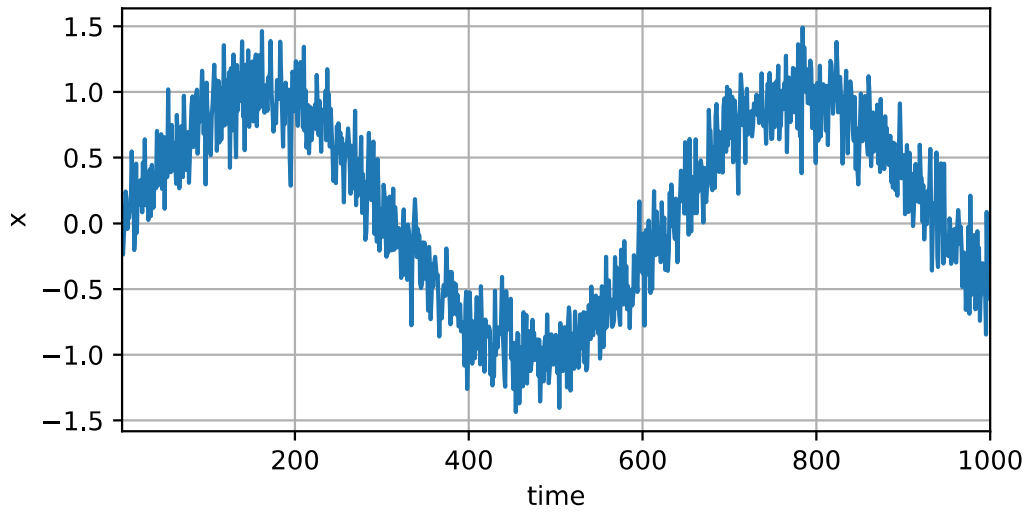
In fact, if we have a Markov model, we can obtain a reverse conditional probability distribution, too. In many cases, however, there exists a natural direction for the data, namely going forward in time. It is clear that future events cannot influence the past. Hence, if we change $x_t$, we may be able to influence what happens for $x_{t+1}$ going forward but not the converse. That is, if we change $x_t$, the distribution over past events will not change. Consequently, it ought to be easier to explain $P(x_{t+1} \mid x_t)$ rather than $P(x_t \mid x_{t+1})$. For instance, it has been shown that in some cases we can find $x_{t+1} = f(x_t) + \epsilon$ for some additive noise $\epsilon$, whereas the converse is not true (Hoyer et al., 2009). This is great news, since it is typically the forward direction that we are interested in estimating. The book by Peters et al. has explained more on this topic (Peters et al., 2017a). We are barely scratching the surface of it.

## 8.1.2 Training

After reviewing so many statistical tools, let us try this out in practice. We begin by generating some data. To keep things simple we generate our sequence data by using a sine function with some additive noise for time steps $1, 2, \ldots, 1000$.

```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
```

```
T = 1000  # Generate a total of 1000 points
time = torch.arange(1, T + 1, dtype=torch.float32)
x = torch.sin(0.01 * time) + torch.normal(0, 0.2, (T,))
d2l.plot(time, [x], 'time', 'x', xlim=[1, 1000], figsize=(6, 3))
```



Next, we need to turn such a sequence into features and labels that our model can train on. Based on the embedding dimension $\tau$ we map the data into pairs $y_t = x_t$ and $\mathbf{x}_t = [x_{t-\tau}, \ldots, x_{t-1}]$. The astute reader might have noticed that this gives us $\tau$ fewer data examples, since we do not have sufficient history for the first $\tau$ of them. A simple fix, in particular if the sequence is long, is to discard those few terms. Alternatively we could pad the sequence with zeros. Here we only use the first 600 feature-label pairs for training.

```
tau = 4
features = torch.zeros((T - tau, tau))
for i in range(tau):
    features[:, i] = x[i: T - tau + i]
labels = x[tau:].reshape((-1, 1))
```

```
batch_size, n_train = 16, 600
# Only the first `n_train` examples are used for training
train_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                            batch_size, is_train=True)
```

Here we keep the architecture fairly simple: just an MLP with two fully-connected layers, ReLU activation, and squared loss.

```
# Function for initializing the weights of the network
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)

# A simple MLP
def get_net():
    net = nn.Sequential(nn.Linear(4, 10),
                        nn.ReLU(),
                        nn.Linear(10, 1))
    net.apply(init_weights)
    return net

# Note: `MSELoss` computes squared error without the 1/2 factor
loss = nn.MSELoss(reduction='none')
```

Now we are ready to train the model. The code below is essentially identical to the training loop in previous sections, such as Section 3.3. Thus, we will not delve into much detail.

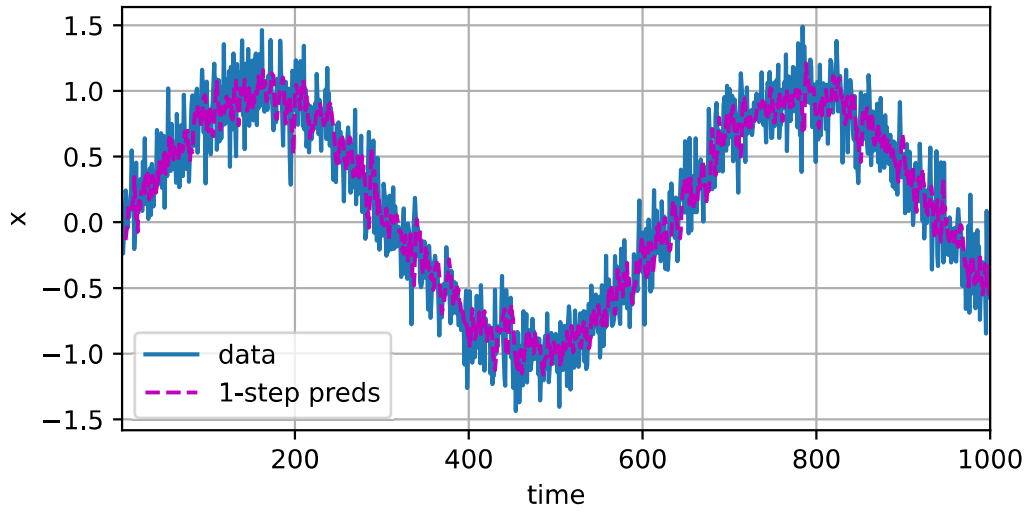
```
def train(net, train_iter, loss, epochs, lr):
    trainer = torch.optim.Adam(net.parameters(), lr)
    for epoch in range(epochs):
        for X, y in train_iter:
            trainer.zero_grad()
            l = loss(net(X), y)
            l.sum().backward()
            trainer.step()
        print(f'epoch {epoch + 1}, '
              f'loss: {d2l.evaluate_loss(net, train_iter, loss):f}')

net = get_net()
train(net, train_iter, loss, 5, 0.01)
```

```
epoch 1, loss: 0.081136
epoch 2, loss: 0.064371
epoch 3, loss: 0.057287
epoch 4, loss: 0.055891
epoch 5, loss: 0.053013
```

### 8.1.3 Prediction

Since the training loss is small, we would expect our model to work well. Let us see what this means in practice. The first thing to check is how well the model is able to predict what happens just in the next time step, namely the *one-step-ahead prediction*.

```
onestep_preds = net(features)
d2l.plot([time, time[tau:]], [x.detach().numpy(), onestep_preds.detach().numpy()], 'time',
         'x', legend=['data', '1-step preds'], xlim=[1, 1000], figsize=(6, 3))
```



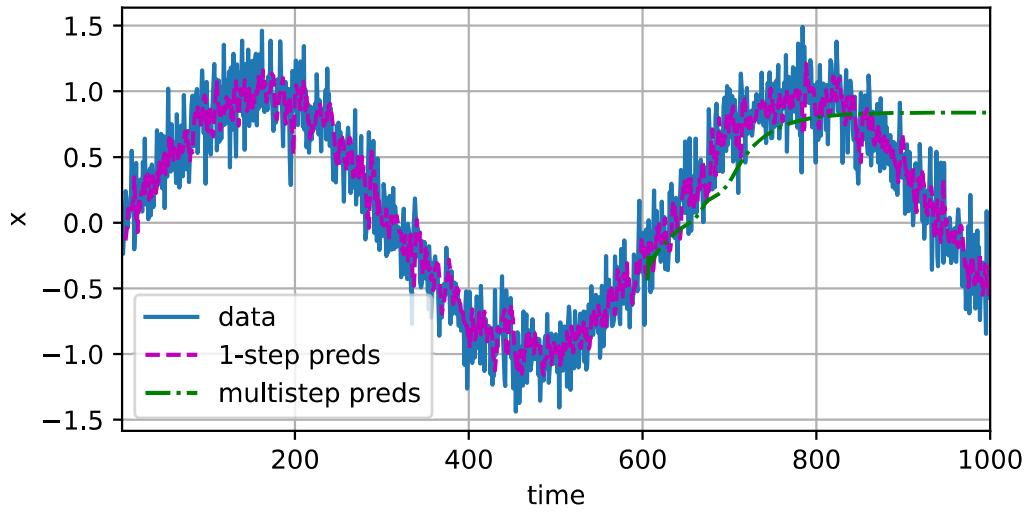

The one-step-ahead predictions look nice, just as we expected. Even beyond 604 (`n_train + tau`) observations the predictions still look trustworthy. However, there is just one little problem to this: if we observe sequence data only until time step 604, we cannot hope to receive the inputs for all the future one-step-ahead predictions. Instead, we need to work our way forward one step at a time:

$$\begin{aligned}
\hat{x}_{605} &= f(x_{601}, x_{602}, x_{603}, x_{604}),\\
\hat{x}_{606} &= f(x_{602}, x_{603}, x_{604}, \hat{x}_{605}),\\
\hat{x}_{607} &= f(x_{603}, x_{604}, \hat{x}_{605}, \hat{x}_{606}),\\
\hat{x}_{608} &= f(x_{604}, \hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}),\\
\hat{x}_{609} &= f(\hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}, \hat{x}_{608}),\\
&\ldots
\end{aligned} \tag{8.1.6}$$

Generally, for an observed sequence up to $x_t$, its predicted output $\hat{x}_{t+k}$ at time step $t+k$ is called the *$k$-step-ahead prediction*. Since we have observed up to $x_{604}$, its $k$-step-ahead prediction is $\hat{x}_{604+k}$. In other words, we will have to use our own predictions to make multistep-ahead predictions. Let us see how well this goes.

```
multistep_preds = torch.zeros(T)
multistep_preds[: n_train + tau] = x[: n_train + tau]
for i in range(n_train + tau, T):
    multistep_preds[i] = net(
        multistep_preds[i - tau:i].reshape((1, -1)))
```

```
d2l.plot([time, time[tau:], time[n_train + tau:]],
         [x.detach().numpy(), onestep_preds.detach().numpy(),
          multistep_preds[n_train + tau:].detach().numpy()], 'time',
         'x', legend=['data', '1-step preds', 'multistep preds'],
         xlim=[1, 1000], figsize=(6, 3))
```




As the above example shows, this is a spectacular failure. The predictions decay to a constant pretty quickly after a few prediction steps. Why did the algorithm work so poorly? This is ultimately due to the fact that the errors build up. Let us say that after step 1 we have some error $\epsilon_1 = \bar{\epsilon}$. Now the *input* for step 2 is perturbed by $\epsilon_1$, hence we suffer some error in the order of $\epsilon_2 = \bar{\epsilon} + c\epsilon_1$ for some constant $c$, and so on. The error can diverge rather rapidly from the true observations. This is a common phenomenon. For instance, weather forecasts for the next 24 hours tend to be pretty accurate but beyond that the accuracy declines rapidly. We will discuss methods for improving this throughout this chapter and beyond.

Let us take a closer look at the difficulties in $k$-step-ahead predictions by computing predictions on the entire sequence for $k = 1, 4, 16, 64$.

```
max_steps = 64
```

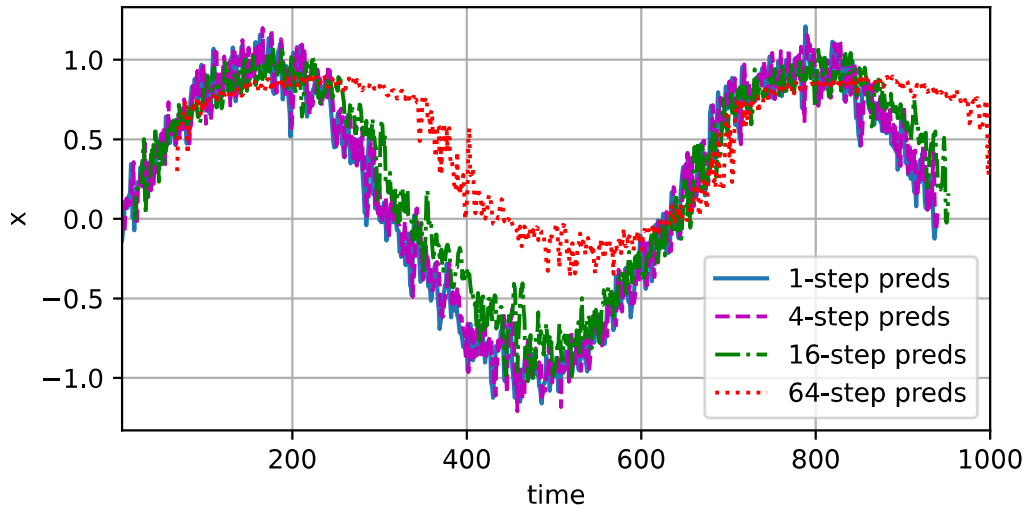
```
features = torch.zeros((T - tau - max_steps + 1, tau + max_steps))
# Column `i` (`i` < `tau`) are observations from `x` for time steps from
# `i + 1` to `i + T - tau - max_steps + 1`
for i in range(tau):
    features[:, i] = x[i: i + T - tau - max_steps + 1]

# Column `i` (`i` >= `tau`) are the (`i - tau + 1`)-step-ahead predictions for
# time steps from `i + 1` to `i + T - tau - max_steps + 1`
```

(continues on next page)

(continued from previous page)

```
for i in range(tau, tau + max_steps):
    features[:, i] = net(features[:, i - tau:i]).reshape(-1)
```

```
steps = (1, 4, 16, 64)
d2l.plot([time[tau + i - 1: T - max_steps + i] for i in steps],
         [features[:, (tau + i - 1)].detach().numpy() for i in steps], 'time', 'x',
         legend=[f'{i}-step preds' for i in steps], xlim=[5, 1000],
         figsize=(6, 3))
```




This clearly illustrates how the quality of the prediction changes as we try to predict further into the future. While the 4-step-ahead predictions still look good, anything beyond that is almost useless.

## Summary

- There is quite a difference in difficulty between interpolation and extrapolation. Consequently, if you have a sequence, always respect the temporal order of the data when training, i.e., never train on future data.
- Sequence models require specialized statistical tools for estimation. Two popular choices are autoregressive models and latent-variable autoregressive models.
- For causal models (e.g., time going forward), estimating the forward direction is typically a lot easier than the reverse direction.
- For an observed sequence up to time step $t$, its predicted output at time step $t + k$ is the $k$-*step-ahead prediction*. As we predict further in time by increasing $k$, the errors accumulate and the quality of the prediction degrades, often dramatically.

### Exercises

1. Improve the model in the experiment of this section.
    1. Incorporate more than the past 4 observations? How many do you really need?
    2. How many past observations would you need if there was no noise? Hint: you can write sin and cos as a differential equation.
    3. Can you incorporate older observations while keeping the total number of features constant? Does this improve accuracy? Why?
    4. Change the neural network architecture and evaluate the performance.
2. An investor wants to find a good security to buy. He looks at past returns to decide which one is likely to do well. What could possibly go wrong with this strategy?
3. Does causality also apply to text? To which extent?
4. Give an example for when a latent autoregressive model might be needed to capture the dynamic of the data.

## 8.2 Text Preprocessing

We have reviewed and evaluated statistical tools and prediction challenges for sequence data. Such data can take many forms. Specifically, as we will focus on in many chapters of the book, 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. To facilitate our future experiments with sequence data, we will dedicate this section to explain common preprocessing steps for text. Usually, these steps are:

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.

```
import collections
import re
from d2l import torch as d2l
```

### 8.2.1 Reading the Dataset

To get started we load text from H. G. Wells' *The Time Machine*[100]. This is a fairly small corpus of just over 30000 words, but for the purpose of what we want to illustrate this is just fine. More realistic document collections contain many billions of words. The following function reads the dataset into a list of text lines, where each line is a string. For simplicity, here we ignore punctuation and capitalization.

```
#@save
d2l.DATA_HUB['time_machine'] = (d2l.DATA_URL + 'timemachine.txt',
                                '090b5e7e70c295757f55df93cb0a180b9691891a')

def read_time_machine():  #@save
    """Load the time machine dataset into a list of text lines."""
    with open(d2l.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])
```

```
Downloading ../data/timemachine.txt from http://d2l-data.s3-accelerate.amazonaws.com/
↪timemachine.txt...
# text lines: 3221
the time machine by h g wells
twinkled and his usually pale face was flushed and animated the
```

### 8.2.2 Tokenization

The following `tokenize` function takes a list (`lines`) as the input, where each element 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.

```
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']
[]
```

(continues on next page)

[100] http://www.gutenberg.org/ebooks/35

(continued from previous page)

```
[]
[]
[]
['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']
```

### 8.2.3 Vocabulary

The string type of the token is inconvenient to be used by models, which take numerical inputs. Now let us build a dictionary, often called *vocabulary* as well, to map string tokens into numerical indices starting from 0. To do so, we first count the unique tokens in all the documents from the training set, namely a *corpus*, and then assign a numerical index to each unique token according to its frequency. 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>". We optionally 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.

```
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[1],
                                   reverse=True)
        # The index for the unknown token is 0
        self.idx_to_token = ['<unk>'] + reserved_tokens
        self.token_to_idx = {token: idx
                             for idx, token in enumerate(self.idx_to_token)}
        for token, freq in self._token_freqs:
            if freq < min_freq:
                break
            if token not in self.token_to_idx:
                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)
```

(continues on next page)

(continued from previous page)

```
        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]

    @property
    def unk(self):  # Index for the unknown token
        return 0

    @property
    def token_freqs(self):  # Index for the unknown token
        return self._token_freqs

def count_corpus(tokens):  #@save
    """Count token frequencies."""
    # Here `tokens` is a 1D list or 2D list
    if len(tokens) == 0 or isinstance(tokens[0], list):
        # Flatten a list of token lists into a list of tokens
        tokens = [token for line in tokens for token in line]
    return collections.Counter(tokens)
```

We construct a vocabulary using the time machine dataset as the corpus. Then we print the first few frequent tokens with their indices.

```
vocab = Vocab(tokens)
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)]
```

Now we can 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, 2183, 2184, 400]
words: ['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and',
↪'animated', 'the']
indices: [2186, 3, 25, 1044, 362, 113, 7, 1421, 3, 1045, 1]
```

### 8.2.4 Putting All Things 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: (i) we tokenize text into characters, not words, to simplify the training in later sections; (ii) `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.

```
def load_corpus_time_machine(max_tokens=-1):  #@save
    """Return token indices and the vocabulary of the time machine dataset."""
    lines = read_time_machine()
    tokens = tokenize(lines, 'char')
    vocab = Vocab(tokens)
    # Since each text line in the time machine dataset is not necessarily a
    # sentence or a paragraph, flatten all the text lines into a single list
    corpus = [vocab[token] for line in tokens for token in line]
    if max_tokens > 0:
        corpus = corpus[:max_tokens]
    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)
```

```
(170580, 28)
```

## Summary

- Text is an important form of sequence data.
- To preprocess text, we usually split text into tokens, build a vocabulary to map token strings into numerical indices, and convert text data into token indices for models to manipulate.

## Exercises

1. Tokenization is a key preprocessing step. It varies for different languages. Try to find another three commonly used methods to tokenize text.
2. In the experiment of this section, tokenize text into words and vary the `min_freq` arguments of the `Vocab` instance. How does this affect the vocabulary size?

## 8.3 Language Models and the Dataset

In Section 8.2, we see how to map text data into tokens, where these tokens can be viewed as a sequence of discrete observations, such as words or characters. Assume that the tokens in a text sequence of length $T$ are in turn $x_1, x_2, \ldots, x_T$. Then, in the text sequence, $x_t (1 \leq t \leq T)$ can be considered as the observation or label at time step $t$. Given such a text sequence, the goal of a *language model* is to estimate the joint probability of the sequence

$$P(x_1, x_2, \ldots, x_T). \tag{8.3.1}$$

Language models are incredibly useful. For instance, an ideal language model would be able to generate natural text just on its own, simply by drawing one token at a time $x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1)$. Quite unlike the monkey using a typewriter, all text emerging from such a model would pass as natural language, e.g., English text. Furthermore, it would be sufficient for generating a meaningful dialog, simply by conditioning the text on previous dialog fragments. Clearly we are still very far from designing such a system, since it would need to *understand* the text rather than just generate grammatically sensible content.

Nonetheless, language models are of great service even in their limited form. For instance, the phrases "to recognize speech" and "to wreck a nice beach" sound very similar. This can cause ambiguity in speech recognition, which is easily resolved through a language model that rejects the second translation as outlandish. Likewise, in a document summarization algorithm it is worthwhile knowing that "dog bites man" is much more frequent than "man bites dog", or that "I want to eat grandma" is a rather disturbing statement, whereas "I want to eat, grandma" is much more benign.

### 8.3.1 Learning a Language Model

The obvious question is how we should model a document, or even a sequence of tokens. Suppose that we tokenize text data at the word level. We can take recourse to the analysis we applied to sequence models in Section 8.1. Let us start by applying basic probability rules:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}). \tag{8.3.2}$$

For example, the probability of a text sequence containing four words would be given as:

$$P(\text{deep}, \text{learning}, \text{is}, \text{fun}) = P(\text{deep})P(\text{learning} \mid \text{deep})P(\text{is} \mid \text{deep}, \text{learning})P(\text{fun} \mid \text{deep}, \text{learning}, \text{is}). \tag{8.3.3}$$

In order to compute the language model, we need to calculate the probability of words and the conditional probability of a word given the previous few words. Such probabilities are essentially language model parameters.

Here, we assume that the training dataset is a large text corpus, such as all Wikipedia entries, Project Gutenberg[102], and all text posted on the Web. The probability of words can be calculated from the relative word frequency of a given word in the training dataset. For example, the estimate $\hat{P}(\text{deep})$ can be calculated as the probability of any sentence starting with the word "deep". A slightly less accurate approach would be to count all occurrences of the word "deep" and divide it

[102] https://en.wikipedia.org/wiki/Project_Gutenberg

by the total number of words in the corpus. This works fairly well, particularly for frequent words. Moving on, we could attempt to estimate

$$\hat{P}(\text{learning} \mid \text{deep}) = \frac{n(\text{deep, learning})}{n(\text{deep})}, \tag{8.3.4}$$

where $n(x)$ and $n(x, x')$ are the number of occurrences of singletons and consecutive word pairs, respectively. Unfortunately, estimating the probability of a word pair is somewhat more difficult, since the occurrences of "deep learning" are a lot less frequent. In particular, for some unusual word combinations it may be tricky to find enough occurrences to get accurate estimates. Things take a turn for the worse for three-word combinations and beyond. There will be many plausible three-word combinations that we likely will not see in our dataset. Unless we provide some solution to assign such word combinations nonzero count, we will not be able to use them in a language model. If the dataset is small or if the words are very rare, we might not find even a single one of them.

A common strategy is to perform some form of *Laplace smoothing*. The solution is to add a small constant to all counts. Denote by $n$ the total number of words in the training set and $m$ the number of unique words. This solution helps with singletons, e.g., via

$$\begin{aligned}
\hat{P}(x) &= \frac{n(x) + \epsilon_1/m}{n + \epsilon_1}, \\
\hat{P}(x' \mid x) &= \frac{n(x, x') + \epsilon_2 \hat{P}(x')}{n(x) + \epsilon_2}, \\
\hat{P}(x'' \mid x, x') &= \frac{n(x, x', x'') + \epsilon_3 \hat{P}(x'')}{n(x, x') + \epsilon_3}.
\end{aligned} \tag{8.3.5}$$

Here $\epsilon_1, \epsilon_2$, and $\epsilon_3$ are hyperparameters. Take $\epsilon_1$ as an example: when $\epsilon_1 = 0$, no smoothing is applied; when $\epsilon_1$ approaches positive infinity, $\hat{P}(x)$ approaches the uniform probability $1/m$. The above is a rather primitive variant of what other techniques can accomplish (Wood et al., 2011).

Unfortunately, models like this get unwieldy rather quickly for the following reasons. First, we need to store all counts. Second, this entirely ignores the meaning of the words. For instance, "cat" and "feline" should occur in related contexts. It is quite difficult to adjust such models to additional contexts, whereas, deep learning based language models are well suited to take this into account. Last, long word sequences are almost certain to be novel, hence a model that simply counts the frequency of previously seen word sequences is bound to perform poorly there.

### 8.3.2 Markov Models and $n$-grams

Before we discuss solutions involving deep learning, we need some more terminology and concepts. Recall our discussion of Markov Models in Section 8.1. Let us apply this to language modeling. A distribution over sequences satisfies the Markov property of first order if $P(x_{t+1} \mid x_t, \ldots, x_1) = P(x_{t+1} \mid x_t)$. Higher orders correspond to longer dependencies. This leads to a number of approximations that we could apply to model a sequence:

$$\begin{aligned}
P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2)P(x_3)P(x_4), \\
P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_2)P(x_4 \mid x_3), \\
P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_1, x_2)P(x_4 \mid x_2, x_3).
\end{aligned} \tag{8.3.6}$$

The probability formulae that involve one, two, and three variables are typically referred to as *unigram*, *bigram*, and *trigram* models, respectively. In the following, we will learn how to design better models.

### 8.3.3 Natural Language Statistics

Let us see how this works on real data. We construct a vocabulary based on the time machine dataset as introduced in Section 8.2 and print the top 10 most frequent words.

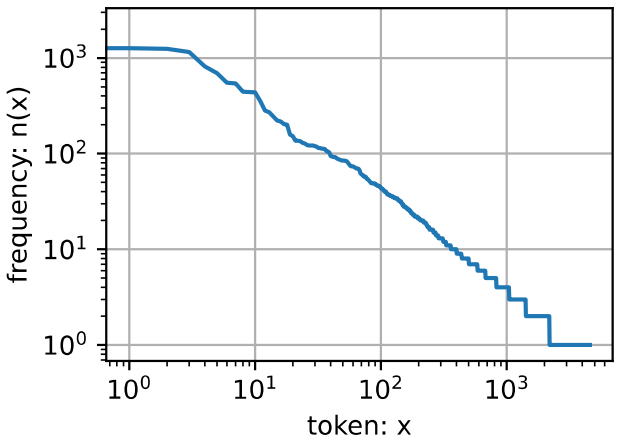
```
import random
import torch
from d2l import torch as d2l
```

```
tokens = d2l.tokenize(d2l.read_time_machine())
# Since each text line is not necessarily a sentence or a paragraph, we
# concatenate all text lines
corpus = [token for line in tokens for token in line]
vocab = d2l.Vocab(corpus)
vocab.token_freqs[:10]
```

```
[('the', 2261),
 ('i', 1267),
 ('and', 1245),
 ('of', 1155),
 ('a', 816),
 ('to', 695),
 ('was', 552),
 ('in', 541),
 ('that', 443),
 ('my', 440)]
```

As we can see, the most popular words are actually quite boring to look at. They are often referred to as *stop words* and thus filtered out. Nonetheless, they still carry meaning and we will still use them. Besides, it is quite clear that the word frequency decays rather rapidly. The $10^{\text{th}}$ most frequent word is less than $1/5$ as common as the most popular one. To get a better idea, we plot the figure of the word frequency.

```
freqs = [freq for token, freq in vocab.token_freqs]
d2l.plot(freqs, xlabel='token: x', ylabel='frequency: n(x)',
         xscale='log', yscale='log')
```



We are on to something quite fundamental here: the word frequency decays rapidly in a well-defined way. After dealing with the first few words as exceptions, all the remaining words roughly follow a straight line on a log-log plot. This means that words satisfy *Zipf's law*, which states that the frequency $n_i$ of the $i^{\text{th}}$ most frequent word is:

$$n_i \propto \frac{1}{i^\alpha}, \tag{8.3.7}$$

which is equivalent to

$$\log n_i = -\alpha \log i + c, \tag{8.3.8}$$

where $\alpha$ is the exponent that characterizes the distribution and $c$ is a constant. This should already give us pause if we want to model words by counting statistics and smoothing. After all, we will significantly overestimate the frequency of the tail, also known as the infrequent words. But what about the other word combinations, such as bigrams, trigrams, and beyond? Let us see whether the bigram frequency behaves in the same manner as the unigram frequency.

```
bigram_tokens = [pair for pair in zip(corpus[:-1], corpus[1:])]
bigram_vocab = d2l.Vocab(bigram_tokens)
bigram_vocab.token_freqs[:10]
```

```
[(('of', 'the'), 309),
 (('in', 'the'), 169),
 (('i', 'had'), 130),
 (('i', 'was'), 112),
 (('and', 'the'), 109),
 (('the', 'time'), 102),
 (('it', 'was'), 99),
 (('to', 'the'), 85),
 (('as', 'i'), 78),
 (('of', 'a'), 73)]
```

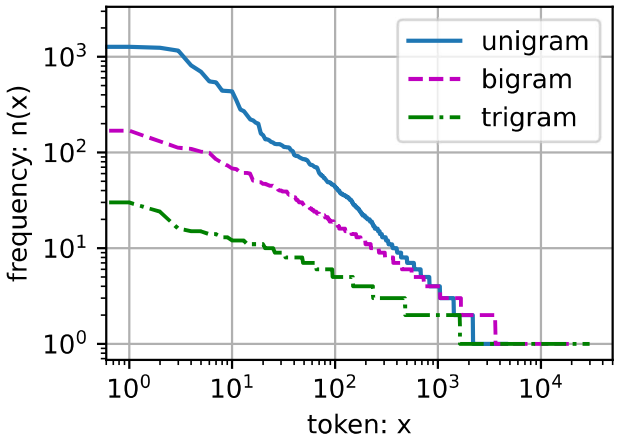
One thing is notable here. Out of the ten most frequent word pairs, nine are composed of both stop words and only one is relevant to the actual book—"the time". Furthermore, let us see whether the trigram frequency behaves in the same manner.

```
trigram_tokens = [triple for triple in zip(
    corpus[:-2], corpus[1:-1], corpus[2:])]
trigram_vocab = d2l.Vocab(trigram_tokens)
trigram_vocab.token_freqs[:10]
```

```
[(('the', 'time', 'traveller'), 59),
 (('the', 'time', 'machine'), 30),
 (('the', 'medical', 'man'), 24),
 (('it', 'seemed', 'to'), 16),
 (('it', 'was', 'a'), 15),
 (('here', 'and', 'there'), 15),
 (('seemed', 'to', 'me'), 14),
 (('i', 'did', 'not'), 14),
 (('i', 'saw', 'the'), 13),
 (('i', 'began', 'to'), 13)]
```

Last, let us visualize the token frequency among these three models: unigrams, bigrams, and trigrams.

```
bigram_freqs = [freq for token, freq in bigram_vocab.token_freqs]
trigram_freqs = [freq for token, freq in trigram_vocab.token_freqs]
d2l.plot([freqs, bigram_freqs, trigram_freqs], xlabel='token: x',
         ylabel='frequency: n(x)', xscale='log', yscale='log',
         legend=['unigram', 'bigram', 'trigram'])
```




This figure is quite exciting for a number of reasons. First, beyond unigram words, sequences of words also appear to be following Zipf's law, albeit with a smaller exponent $\alpha$ in (8.3.7), depending on the sequence length. Second, the number of distinct $n$-grams is not that large. This gives us hope that there is quite a lot of structure in language. Third, many $n$-grams occur very rarely, which makes Laplace smoothing rather unsuitable for language modeling. Instead, we will use deep learning based models.

### 8.3.4 Reading Long Sequence Data

Since sequence data are by their very nature sequential, we need to address the issue of processing it. We did so in a rather ad-hoc manner in Section 8.1. When sequences get too long to be processed by models all at once, we may wish to split such sequences for reading. Now let us describe general strategies. Before introducing the model, let us assume that we will use a neural network to train a language model, where the network processes a minibatch of sequences with predefined length, say $n$ time steps, at a time. Now the question is how to read minibatches of features and labels at random.

To begin with, since a text sequence can be arbitrarily long, such as the entire *The Time Machine* book, we can partition such a long sequence 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. Fig. 8.3.1 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. Note that we have quite some freedom since we could pick an arbitrary offset that indicates the initial position.

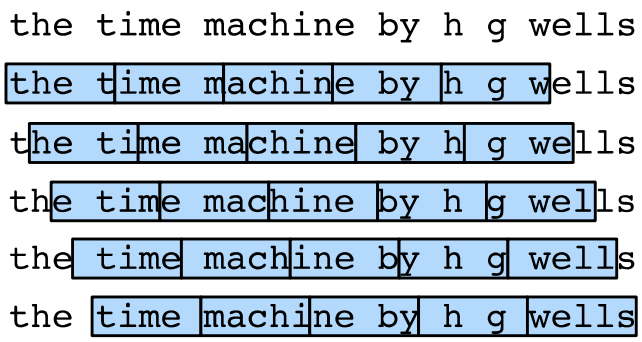



Fig. 8.3.1: Different offsets lead to different subsequences when splitting up text.

Hence, which one should we pick from Fig. 8.3.1? In fact, all of them are equally good. However, if we pick just one offset, there is limited coverage of all the possible subsequences for training our network. Therefore, we can start with a random offset to partition a sequence to get both *coverage* and *randomness*. In the following, we describe how to accomplish this for both *random sampling* and *sequential partitioning* strategies.

### Random Sampling

In random sampling, each example is a subsequence arbitrarily captured on the original long sequence. The subsequences from two adjacent random minibatches during iteration are not necessarily adjacent on the original sequence. For language modeling, the target is to predict the next token based on what tokens we have seen so far, hence the labels are the original sequence, shifted by one token.

The following code randomly generates a minibatch from the data each time. Here, the argument `batch_size` specifies the number of subsequence examples in each minibatch and `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 (inclusive of `num_steps - 1`) to partition a
    # sequence
    corpus = corpus[random.randint(0, num_steps - 1):]
    # 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_batches = num_subseqs // batch_size
    for i in range(0, batch_size * num_batches, batch_size):
```

(continues on next page)

(continued from previous page)

```
        # 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)
```

Let us manually generate a sequence from 0 to 34. We assume that the batch size and numbers of time steps are 2 and 5, respectively. 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.

```
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([[24, 25, 26, 27, 28],
        [ 9, 10, 11, 12, 13]])
Y: tensor([[25, 26, 27, 28, 29],
        [10, 11, 12, 13, 14]])
X:  tensor([[29, 30, 31, 32, 33],
        [ 4,  5,  6,  7,  8]])
Y: tensor([[30, 31, 32, 33, 34],
        [ 5,  6,  7,  8,  9]])
X:  tensor([[14, 15, 16, 17, 18],
        [19, 20, 21, 22, 23]])
Y: tensor([[15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24]])
```

### Sequential Partitioning

In addition to random sampling of the original sequence, we can also ensure that the subsequences from two adjacent minibatches during iteration are adjacent on the original sequence. This strategy preserves the order of split subsequences when iterating over minibatches, hence is called sequential partitioning.

```
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_steps * num_batches, num_steps):
        X = Xs[:, i: i + num_steps]
        Y = Ys[:, i: i + num_steps]
        yield X, Y
```

Using the same settings, let us print features `X` and labels `Y` for each minibatch of subsequences read by sequential partitioning. Note that the subsequences from two adjacent minibatches dur-

ing iteration are indeed adjacent on the original sequence.

```
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]])
```

Now we wrap the above two sampling functions to a class so that we can use it as a data iterator later.

```
class SeqDataLoader:  #@save
    """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 = d2l.seq_data_iter_random
        else:
            self.data_iter_fn = d2l.seq_data_iter_sequential
        self.corpus, self.vocab = d2l.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)
```

Last, we define a function `load_data_time_machine` that returns both the data iterator and the vocabulary, so we can use it similarly as other other functions with the `load_data` prefix, such as `d2l.load_data_fashion_mnist` defined in Section 3.5.

```
def load_data_time_machine(batch_size, num_steps,  #@save
                           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
```

### Summary

- Language models are key to natural language processing.
- $n$-grams provide a convenient model for dealing with long sequences by truncating the dependence.
- Long sequences suffer from the problem that they occur very rarely or never.
- Zipf's law governs the word distribution for not only unigrams but also the other $n$-grams.
- There is a lot of structure but not enough frequency to deal with infrequent word combinations efficiently via Laplace smoothing.
- The main choices for reading long sequences are random sampling and sequential partitioning. The latter can ensure that the subsequences from two adjacent minibatches during iteration are adjacent on the original sequence.

### Exercises

1. Suppose there are $100,000$ words in the training dataset. How much word frequency and multi-word adjacent frequency does a four-gram need to store?
2. How would you model a dialogue?
3. Estimate the exponent of Zipf's law for unigrams, bigrams, and trigrams.
4. What other methods can you think of for reading long sequence data?
5. Consider the random offset that we use for reading long sequences.
    1. Why is it a good idea to have a random offset?
    2. Does it really lead to a perfectly uniform distribution over the sequences on the document?
    3. What would you have to do to make things even more uniform?
6. If we want a sequence example to be a complete sentence, what kind of problem does this introduce in minibatch sampling? How can we fix the problem?

## 8.4 Recurrent Neural Networks

In Section 8.3 we introduced $n$-gram models, where the conditional probability of word $x_t$ at time step $t$ only depends on the $n$ _ 1 previous words. If we want to incorporate the possible effect of words earlier than time step $t-(n-1)$ on $x_t$, we need to increase $n$. However, the number of model parameters would also increase exponentially with it, as we need to store $|\mathcal{V}|^n$ numbers for a vocabulary set $\mathcal{V}$. Hence, rather than modeling $P(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$ it is preferable to use a latent variable model:

$$P(x_t \mid x_{t-1}, \ldots, x_1) \approx P(x_t \mid h_{t-1}), \tag{8.4.1}$$

where $h_{t-1}$ is a *hidden state* (also known as a hidden variable) that stores the sequence information up to time step $t-1$. In general, the hidden state at any time step $t$ could be computed based on both the current input $x_t$ and the previous hidden state $h_{t-1}$:

$$h_t = f(x_t, h_{t-1}). \tag{8.4.2}$$

For a sufficiently powerful function $f$ in (8.4.2), the latent variable model is not an approximation. After all, $h_t$ may simply store all the data it has observed so far. However, it could potentially make both computation and storage expensive.

Recall that we have discussed hidden layers with hidden units in Chapter 4. It is noteworthy that hidden layers and hidden states refer to two very different concepts. Hidden layers are, as explained, layers that are hidden from view on the path from input to output. Hidden states are technically speaking *inputs* to whatever we do at a given step, and they can only be computed by looking at data at previous time steps.

*Recurrent neural networks* (RNNs) are neural networks with hidden states. Before introducing the RNN model, we first revisit the MLP model introduced in Section 4.1.

### 8.4.1 Neural Networks without Hidden States

Let us take a look at an MLP with a single hidden layer. Let the hidden layer's activation function be $\phi$. Given a minibatch of examples $\mathbf{X} \in \mathbb{R}^{n \times d}$ with batch size $n$ and $d$ inputs, the hidden layer's output $\mathbf{H} \in \mathbb{R}^{n \times h}$ is calculated as

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h). \tag{8.4.3}$$

In (8.4.3), we have the weight parameter $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, the bias parameter $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$, and the number of hidden units $h$, for the hidden layer. Thus, broadcasting (see Section 2.1.3) is applied during the summation. Next, the hidden variable $\mathbf{H}$ is used as the input of the output layer. The output layer is given by

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q, \tag{8.4.4}$$

where $\mathbf{O} \in \mathbb{R}^{n \times q}$ is the output variable, $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ is the weight parameter, and $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ is the bias parameter of the output layer. If it is a classification problem, we can use $\text{softmax}(\mathbf{O})$ to compute the probability distribution of the output categories.

This is entirely analogous to the regression problem we solved previously in Section 8.1, hence we omit details. Suffice it to say that we can pick feature-label pairs at random and learn the parameters of our network via automatic differentiation and stochastic gradient descent.

### 8.4.2 Recurrent Neural Networks with Hidden States

Matters are entirely different when we have hidden states. Let us look at the structure in some more detail.

Assume that we have a minibatch of inputs $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ at time step $t$. In other words, for a minibatch of $n$ sequence examples, each row of $\mathbf{X}_t$ corresponds to one example at time step $t$ from the sequence. Next, denote by $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ the hidden variable of time step $t$. Unlike the MLP, here we save the hidden variable $\mathbf{H}_{t-1}$ from the previous time step and introduce a new weight parameter $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ to describe how to use the hidden variable of the previous time step in the

current time step. Specifically, the calculation of the hidden variable of the current time step is determined by the input of the current time step together with the hidden variable of the previous time step:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h). \tag{8.4.5}$$

Compared with (8.4.3), (8.4.5) adds one more term $\mathbf{H}_{t-1}\mathbf{W}_{hh}$ and thus instantiates (8.4.2). From the relationship between hidden variables $\mathbf{H}_t$ and $\mathbf{H}_{t-1}$ of adjacent time steps, we know that these variables captured and retained the sequence's historical information up to their current time step, just like the state or memory of the neural network's current time step. Therefore, such a hidden variable is called a *hidden state*. Since the hidden state uses the same definition of the previous time step in the current time step, the computation of (8.4.5) is *recurrent*. Hence, neural networks with hidden states based on recurrent computation are named *recurrent neural networks*. Layers that perform the computation of (8.4.5) in RNNs are called *recurrent layers*.

There are many different ways for constructing RNNs. RNNs with a hidden state defined by (8.4.5) are very common. For time step $t$, the output of the output layer is similar to the computation in the MLP:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \tag{8.4.6}$$

Parameters of the RNN include the weights $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$, and the bias $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ of the hidden layer, together with the weights $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ and the bias $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ of the output layer. It is worth mentioning that even at different time steps, RNNs always use these model parameters. Therefore, the parameterization cost of an RNN does not grow as the number of time steps increases.

Fig. 8.4.1 illustrates the computational logic of an RNN at three adjacent time steps. At any time step $t$, the computation of the hidden state can be treated as: (i) concatenating the input $\mathbf{X}_t$ at the current time step $t$ and the hidden state $\mathbf{H}_{t-1}$ at the previous time step $t-1$; (ii) feeding the concatenation result into a fully-connected layer with the activation function $\phi$. The output of such a fully-connected layer is the hidden state $\mathbf{H}_t$ of the current time step $t$. In this case, the model parameters are the concatenation of $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$, and a bias of $\mathbf{b}_h$, all from (8.4.5). The hidden state of the current time step $t$, $\mathbf{H}_t$, will participate in computing the hidden state $\mathbf{H}_{t+1}$ of the next time step $t+1$. What is more, $\mathbf{H}_t$ will also be fed into the fully-connected output layer to compute the output $\mathbf{O}_t$ of the current time step $t$.

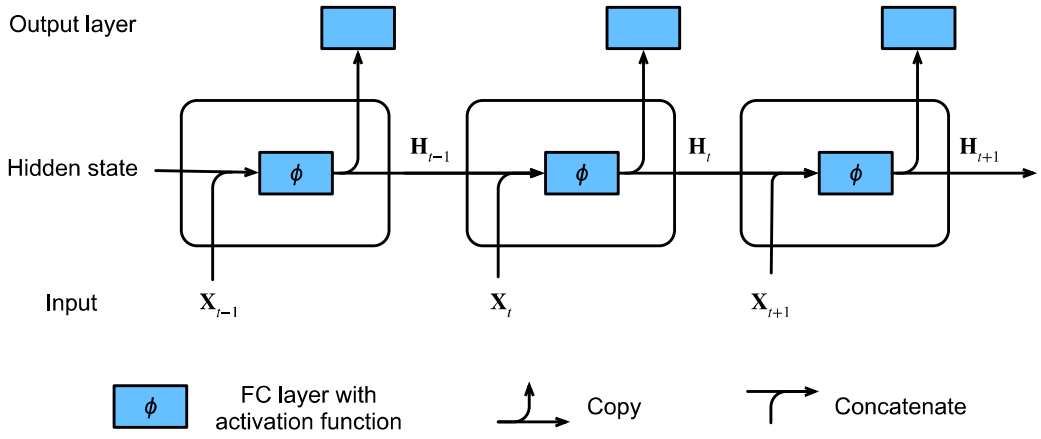



Fig. 8.4.1: An RNN with a hidden state.

We just mentioned that the calculation of $\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh}$ for the hidden state is equivalent to matrix multiplication of concatenation of $\mathbf{X}_t$ and $\mathbf{H}_{t-1}$ and concatenation of $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$. Though this can be proven in mathematics, in the following we just use a simple code snippet to show this. To begin with, we define matrices `X`, `W_xh`, `H`, and `W_hh`, whose shapes are (3, 1), (1, 4), (3, 4), and (4, 4), respectively. Multiplying `X` by `W_xh`, and `H` by `W_hh`, respectively, and then adding these two multiplications, we obtain a matrix of shape (3, 4).

```
import torch
from d2l import torch as d2l
```

```
X, W_xh = torch.normal(0, 1, (3, 1)), torch.normal(0, 1, (1, 4))
H, W_hh = torch.normal(0, 1, (3, 4)), torch.normal(0, 1, (4, 4))
torch.matmul(X, W_xh) + torch.matmul(H, W_hh)
```

```
tensor([[ 1.4019, -1.5276,  4.5853, -0.2790],
        [-4.3955, -2.4508, -0.4531,  1.3304],
        [-0.3181,  1.3874, -1.7398, -1.2179]])
```

Now we concatenate the matrices `X` and `H` along columns (axis 1), and the matrices `W_xh` and `W_hh` along rows (axis 0). These two concatenations result in matrices of shape (3, 5) and of shape (5, 4), respectively. Multiplying these two concatenated matrices, we obtain the same output matrix of shape (3, 4) as above.

```
torch.matmul(torch.cat((X, H), 1), torch.cat((W_xh, W_hh), 0))
```

```
tensor([[ 1.4019, -1.5276,  4.5853, -0.2790],
        [-4.3955, -2.4508, -0.4531,  1.3304],
        [-0.3181,  1.3874, -1.7398, -1.2179]])
```

### 8.4.3 RNN-based Character-Level Language Models

Recall that for language modeling in Section 8.3, we aim to predict the next token based on the current and past tokens, thus we shift the original sequence by one token as the labels. Bengio et al. first proposed to use a neural network for language modeling (Bengio et al., 2003). In the following we illustrate how RNNs can be used to build a language model. Let the minibatch size be one, and the sequence of the text be "machine". To simplify training in subsequent sections, we tokenize text into characters rather than words and consider a *character-level language model*. Fig. 8.4.2 demonstrates how to predict the next character based on the current and previous characters via an RNN for character-level language modeling.

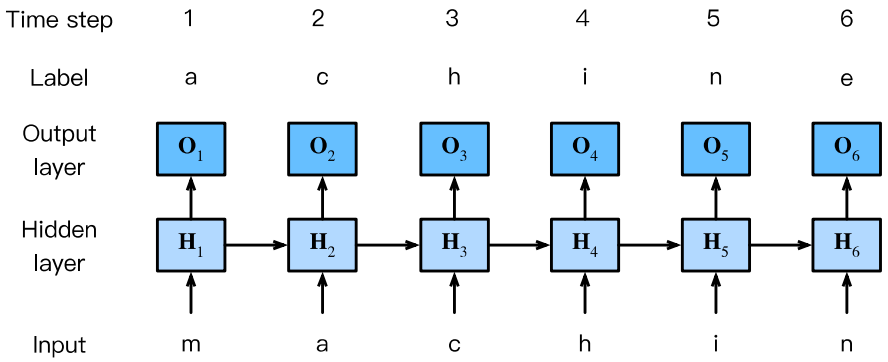



Fig. 8.4.2: A character-level language model based on the RNN. The input and label sequences are "machin" and "achine", respectively.

During the training process, we run a softmax operation on the output from the output layer for each time step, and then use the cross-entropy loss to compute the error between the model output and the label. Due to the recurrent computation of the hidden state in the hidden layer, the output of time step 3 in Fig. 8.4.2, $\mathbf{O}_3$, is determined by the text sequence "m", "a", and "c". Since the next character of the sequence in the training data is "h", the loss of time step 3 will depend on the probability distribution of the next character generated based on the feature sequence "m", "a", "c" and the label "h" of this time step.

In practice, each token is represented by a $d$-dimensional vector, and we use a batch size $n > 1$. Therefore, the input $\mathbf{X}_t$ at time step $t$ will be a $n \times d$ matrix, which is identical to what we discussed in Section 8.4.2.

### 8.4.4 Perplexity

Last, let us discuss about how to measure the language model quality, which will be used to evaluate our RNN-based models in the subsequent sections. One way is to check how surprising the text is. A good language model is able to predict with high-accuracy tokens that what we will see next. Consider the following continuations of the phrase "It is raining", as proposed by different language models:

1. "It is raining outside"
2. "It is raining banana tree"
3. "It is raining piouw;kcj pwepoiut"

In terms of quality, example 1 is clearly the best. The words are sensible and logically coherent. While it might not quite accurately reflect which word follows semantically ("in San Francisco" and "in winter" would have been perfectly reasonable extensions), the model is able to capture which kind of word follows. Example 2 is considerably worse by producing a nonsensical extension. Nonetheless, at least the model has learned how to spell words and some degree of correlation between words. Last, example 3 indicates a poorly trained model that does not fit data properly.

We might measure the quality of the model by computing the likelihood of the sequence. Unfortunately this is a number that is hard to understand and difficult to compare. After all, shorter sequences are much more likely to occur than the longer ones, hence evaluating the model on Tolstoy's magnum opus *War and Peace* will inevitably produce a much smaller likelihood than, say, on Saint-Exupery's novella *The Little Prince*. What is missing is the equivalent of an average.

Information theory comes handy here. We have defined entropy, surprisal, and cross-entropy when we introduced the softmax regression (Section 3.4.7) and more of information theory is discussed in the online appendix on information theory. If we want to compress text, we can ask about predicting the next token given the current set of tokens. A better language model should allow us to predict the next token more accurately. Thus, it should allow us to spend fewer bits in compressing the sequence. So we can measure it by the cross-entropy loss averaged over all the $n$ tokens of a sequence:

$$\frac{1}{n}\sum_{t=1}^{n} -\log P(x_t \mid x_{t-1}, \ldots, x_1), \tag{8.4.7}$$

where $P$ is given by a language model and $x_t$ is the actual token observed at time step $t$ from the sequence. This makes the performance on documents of different lengths comparable. For historical reasons, scientists in natural language processing prefer to use a quantity called *perplexity*. In a nutshell, it is the exponential of (8.4.7):

$$\exp\left(-\frac{1}{n}\sum_{t=1}^{n} \log P(x_t \mid x_{t-1}, \ldots, x_1)\right). \tag{8.4.8}$$

Perplexity can be best understood as the harmonic mean of the number of real choices that we have when deciding which token to pick next. Let us look at a number of cases:

- In the best case scenario, the model always perfectly estimates the probability of the label token as 1. In this case the perplexity of the model is 1.
- In the worst case scenario, the model always predicts the probability of the label token as 0. In this situation, the perplexity is positive infinity.
- At the baseline, the model predicts a uniform distribution over all the available tokens of the vocabulary. In this case, the perplexity equals the number of unique tokens of the vocabulary. In fact, if we were to store the sequence without any compression, this would be the best we could do to encode it. Hence, this provides a nontrivial upper bound that any useful model must beat.

In the following sections, we will implement RNNs for character-level language models and use perplexity to evaluate such models.

## Summary

- A neural network that uses recurrent computation for hidden states is called a recurrent neural network (RNN).
- The hidden state of an RNN can capture historical information of the sequence up to the current time step.
- The number of RNN model parameters does not grow as the number of time steps increases.
- We can create character-level language models using an RNN.
- We can use perplexity to evaluate the quality of language models.

### Exercises

1. If we use an RNN to predict the next character in a text sequence, what is the required dimension for any output?
2. Why can RNNs express the conditional probability of a token at some time step based on all the previous tokens in the text sequence?
3. What happens to the gradient if you backpropagate through a long sequence?
4. What are some of the problems associated with the language model described in this section?

## 8.5 Implementation of Recurrent Neural Networks from Scratch

In this section we will implement an RNN from scratch for a character-level language model, according to our descriptions in Section 8.4. Such a model will be trained on H. G. Wells' *The Time Machine*. As before, we start by reading the dataset first, which is introduced in Section 8.3.

```
%matplotlib inline
import math
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

```
batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

### 8.5.1 One-Hot Encoding

Recall that each token is represented as a numerical index in `train_iter`. Feeding these indices directly to a neural network might make it hard to learn. We often represent each token as a more expressive feature vector. The easiest representation is called *one-hot encoding*, which is introduced in Section 3.4.1.

In a nutshell, we map each index to a different unit vector: assume that the number of different tokens in the vocabulary is $N$ (`len(vocab)`) and the token indices range from 0 to $N-1$. If the index of a token is the integer $i$, then we create a vector of all 0s with a length of $N$ and set the element at position $i$ to 1. This vector is the one-hot vector of the original token. The one-hot vectors with indices 0 and 2 are shown below.

```
F.one_hot(torch.tensor([0, 2]), len(vocab))
```

```
tensor([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0]])
```

The shape of the minibatch that we sample each time is (batch size, number of time steps). The one_hot function transforms such a minibatch into a three-dimensional tensor with the last dimension equals to the vocabulary size (len(vocab)). We often transpose the input so that we will obtain an output of shape (number of time steps, batch size, vocabulary size). This will allow us to more conveniently loop through the outermost dimension for updating hidden states of a minibatch, time step by time step.

```
X = torch.arange(10).reshape((2, 5))
F.one_hot(X.T, 28).shape
```

```
torch.Size([5, 2, 28])
```

### 8.5.2 Initializing the Model Parameters

Next, we initialize the model parameters for the RNN model. The number of hidden units num_hiddens is a tunable hyperparameter. When training language models, the inputs and outputs are from the same vocabulary. Hence, they have the same dimension, which is equal to the vocabulary size.

```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device) * 0.01

    # Hidden layer parameters
    W_xh = normal((num_inputs, num_hiddens))
    W_hh = normal((num_hiddens, num_hiddens))
    b_h = torch.zeros(num_hiddens, device=device)
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    # Attach gradients
    params = [W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

### 8.5.3 RNN Model

To define an RNN model, we first need an `init_rnn_state` function to return the hidden state at initialization. It returns a tensor filled with 0 and with a shape of (batch size, number of hidden units). Using tuples makes it easier to handle situations where the hidden state contains multiple variables, which we will encounter in later sections.

```
def init_rnn_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device), )
```

The following `rnn` function defines how to compute the hidden state and output at a time step. Note that the RNN model loops through the outermost dimension of `inputs` so that it updates hidden states `H` of a minibatch, time step by time step. Besides, the activation function here uses the tanh function. As described in Section 4.1, the mean value of the tanh function is 0, when the elements are uniformly distributed over the real numbers.

```
def rnn(inputs, state, params):
    # Here `inputs` shape: (`num_steps`, `batch_size`, `vocab_size`)
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    # Shape of `X`: (`batch_size`, `vocab_size`)
    for X in inputs:
        H = torch.tanh(torch.mm(X, W_xh) + torch.mm(H, W_hh) + b_h)
        Y = torch.mm(H, W_hq) + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H,)
```

With all the needed functions being defined, next we create a class to wrap these functions and store parameters for an RNN model implemented from scratch.

```
class RNNModelScratch: #@save
    """A RNN Model implemented from scratch."""
    def __init__(self, vocab_size, num_hiddens, device,
                 get_params, init_state, forward_fn):
        self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
        self.params = get_params(vocab_size, num_hiddens, device)
        self.init_state, self.forward_fn = init_state, forward_fn

    def __call__(self, X, state):
        X = F.one_hot(X.T, self.vocab_size).type(torch.float32)
        return self.forward_fn(X, state, self.params)

    def begin_state(self, batch_size, device):
        return self.init_state(batch_size, self.num_hiddens, device)
```

Let us check whether the outputs have the correct shapes, e.g., to ensure that the dimensionality of the hidden state remains unchanged.

```
num_hiddens = 512
net = RNNModelScratch(len(vocab), num_hiddens, d2l.try_gpu(), get_params,
                      init_rnn_state, rnn)
state = net.begin_state(X.shape[0], d2l.try_gpu())
```

(continues on next page)

(continued from previous page)

```
Y, new_state = net(X.to(d2l.try_gpu()), state)
Y.shape, len(new_state), new_state[0].shape
```

```
(torch.Size([10, 28]), 1, torch.Size([2, 512]))
```

We can see that the output shape is (number of time steps $\times$ batch size, vocabulary size), while the hidden state shape remains the same, i.e., (batch size, number of hidden units).

### 8.5.4 Prediction

Let us first define the prediction function to generate new characters following the user-provided `prefix`, which is a string containing several characters. When looping through these beginning characters in `prefix`, we keep passing the hidden state to the next time step without generating any output. This is called the *warm-up* period, during which the model updates itself (e.g., update the hidden state) but does not make predictions. After the warm-up period, the hidden state is generally better than its initialized value at the beginning. So we generate the predicted characters and emit them.

```
def predict_ch8(prefix, num_preds, net, vocab, device):  #@save
    """Generate new characters following the `prefix`."""
    state = net.begin_state(batch_size=1, device=device)
    outputs = [vocab[prefix[0]]]
    get_input = lambda: torch.tensor([outputs[-1]], device=device).reshape((1, 1))
    for y in prefix[1:]:  # Warm-up period
        _, state = net(get_input(), state)
        outputs.append(vocab[y])
    for _ in range(num_preds):  # Predict `num_preds` steps
        y, state = net(get_input(), state)
        outputs.append(int(y.argmax(dim=1).reshape(1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

Now we can test the `predict_ch8` function. We specify the prefix as `time traveller` and have it generate 10 additional characters. Given that we have not trained the network, it will generate nonsensical predictions.

```
predict_ch8('time traveller ', 10, net, vocab, d2l.try_gpu())
```

```
'time traveller gqsjl<unk>wzl<unk>'
```

### 8.5.5 Gradient Clipping

For a sequence of length $T$, we compute the gradients over these $T$ time steps in an iteration, which results in a chain of matrix-products with length $\mathcal{O}(T)$ during backpropagation. As mentioned in Section 4.8, it might result in numerical instability, e.g., the gradients may either explode or vanish, when $T$ is large. Therefore, RNN models often need extra help to stabilize the training.

Generally speaking, when solving an optimization problem, we take update steps for the model parameter, say in the vector form $\mathbf{x}$, in the direction of the negative gradient $\mathbf{g}$ on a minibatch. For

example, with $\eta > 0$ as the learning rate, in one iteration we update $\mathbf{x}$ as $\mathbf{x} - \eta\mathbf{g}$. Let us further assume that the objective function $f$ is well behaved, say, *Lipschitz continuous* with constant $L$. That is to say, for any $\mathbf{x}$ and $\mathbf{y}$ we have

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L\|\mathbf{x} - \mathbf{y}\|. \tag{8.5.1}$$

In this case we can safely assume that if we update the parameter vector by $\eta\mathbf{g}$, then

$$|f(\mathbf{x}) - f(\mathbf{x} - \eta\mathbf{g})| \leq L\eta\|\mathbf{g}\|, \tag{8.5.2}$$

which means that we will not observe a change by more than $L\eta\|\mathbf{g}\|$. This is both a curse and a blessing. On the curse side, it limits the speed of making progress; whereas on the blessing side, it limits the extent to which things can go wrong if we move in the wrong direction.

Sometimes the gradients can be quite large and the optimization algorithm may fail to converge. We could address this by reducing the learning rate $\eta$. But what if we only *rarely* get large gradients? In this case such an approach may appear entirely unwarranted. One popular alternative is to clip the gradient $\mathbf{g}$ by projecting them back to a ball of a given radius, say $\theta$ via

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}. \tag{8.5.3}$$

By doing so we know that the gradient norm never exceeds $\theta$ and that the updated gradient is entirely aligned with the original direction of $\mathbf{g}$. It also has the desirable side-effect of limiting the influence any given minibatch (and within it any given sample) can exert on the parameter vector. This bestows a certain degree of robustness to the model. Gradient clipping provides a quick fix to the gradient exploding. While it does not entirely solve the problem, it is one of the many techniques to alleviate it.

Below we define a function to clip the gradients of a model that is implemented from scratch or a model constructed by the high-level APIs. Also note that we compute the gradient norm over all the model parameters.

```
def grad_clipping(net, theta):  #@save
    """Clip the gradient."""
    if isinstance(net, nn.Module):
        params = [p for p in net.parameters() if p.requires_grad]
    else:
        params = net.params
    norm = torch.sqrt(sum(torch.sum((p.grad ** 2)) for p in params))
    if norm > theta:
        for param in params:
            param.grad[:] *= theta / norm
```

### 8.5.6 Training

Before training the model, let us define a function to train the model in one epoch. It differs from how we train the model of Section 3.6 in three places:

1. Different sampling methods for sequential data (random sampling and sequential partitioning) will result in differences in the initialization of hidden states.
2. We clip the gradients before updating the model parameters. This ensures that the model does not diverge even when gradients blow up at some point during the training process.

3. We use perplexity to evaluate the model. As discussed in Section 8.4.4, this ensures that sequences of different length are comparable.

Specifically, when sequential partitioning is used, we initialize the hidden state only at the beginning of each epoch. Since the $i^{\text{th}}$ subsequence example in the next minibatch is adjacent to the current $i^{\text{th}}$ subsequence example, the hidden state at the end of the current minibatch will be used to initialize the hidden state at the beginning of the next minibatch. In this way, historical information of the sequence stored in the hidden state might flow over adjacent subsequences within an epoch. However, the computation of the hidden state at any point depends on all the previous minibatches in the same epoch, which complicates the gradient computation. To reduce computational cost, we detach the gradient before processing any minibatch so that the gradient computation of the hidden state is always limited to the time steps in one minibatch.

When using the random sampling, we need to re-initialize the hidden state for each iteration since each example is sampled with a random position. Same as the `train_epoch_ch3` function in Section 3.6, `updater` is a general function to update the model parameters. It can be either the `d2l.sgd` function implemented from scratch or the built-in optimization function in a deep learning framework.

```
#@save
def train_epoch_ch8(net, train_iter, loss, updater, device, use_random_iter):
    """Train a net within one epoch (defined in Chapter 8)."""
    state, timer = None, d2l.Timer()
    metric = d2l.Accumulator(2)  # Sum of training loss, no. of tokens
    for X, Y in train_iter:
        if state is None or use_random_iter:
            # Initialize `state` when either it is the first iteration or
            # using random sampling
            state = net.begin_state(batch_size=X.shape[0], device=device)
        else:
            if isinstance(net, nn.Module) and not isinstance(state, tuple):
                # `state` is a tensor for `nn.GRU`
                state.detach_()
            else:
                # `state` is a tuple of tensors for `nn.LSTM` and
                # for our custom scratch implementation
                for s in state:
                    s.detach_()
        y = Y.T.reshape(-1)
        X, y = X.to(device), y.to(device)
        y_hat, state = net(X, state)
        l = loss(y_hat, y.long()).mean()
        if isinstance(updater, torch.optim.Optimizer):
            updater.zero_grad()
            l.backward()
            grad_clipping(net, 1)
            updater.step()
        else:
            l.backward()
            grad_clipping(net, 1)
            # Since the `mean` function has been invoked
            updater(batch_size=1)
        metric.add(l * y.numel(), y.numel())
    return math.exp(metric[0] / metric[1]), metric[1] / timer.stop()
```

The training function supports an RNN model implemented either from scratch or using high-

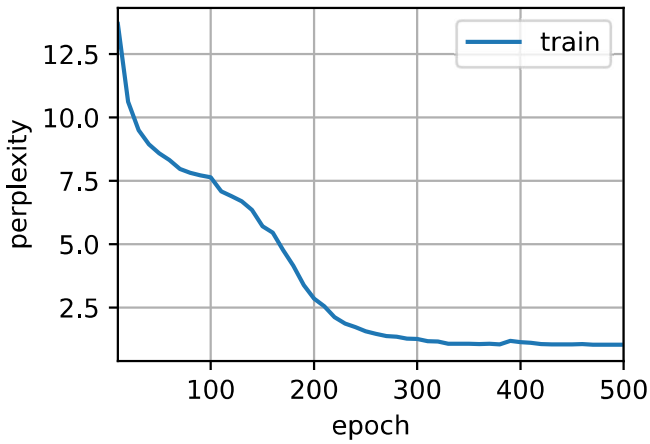
level APIs.

```
#@save
def train_ch8(net, train_iter, vocab, lr, num_epochs, device,
              use_random_iter=False):
    """Train a model (defined in Chapter 8)."""
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='perplexity',
                            legend=['train'], xlim=[10, num_epochs])
    # Initialize
    if isinstance(net, nn.Module):
        updater = torch.optim.SGD(net.parameters(), lr)
    else:
        updater = lambda batch_size: d2l.sgd(net.params, lr, batch_size)
    predict = lambda prefix: predict_ch8(prefix, 50, net, vocab, device)
    # Train and predict
    for epoch in range(num_epochs):
        ppl, speed = train_epoch_ch8(
            net, train_iter, loss, updater, device, use_random_iter)
        if (epoch + 1) % 10 == 0:
            print(predict('time traveller'))
            animator.add(epoch + 1, [ppl])
    print(f'perplexity {ppl:.1f}, {speed:.1f} tokens/sec on {str(device)}')
    print(predict('time traveller'))
    print(predict('traveller'))
```

Now we can train the RNN model. Since we only use 10000 tokens in the dataset, the model needs more epochs to converge better.

```
num_epochs, lr = 500, 1
train_ch8(net, train_iter, vocab, lr, num_epochs, d2l.try_gpu())
```

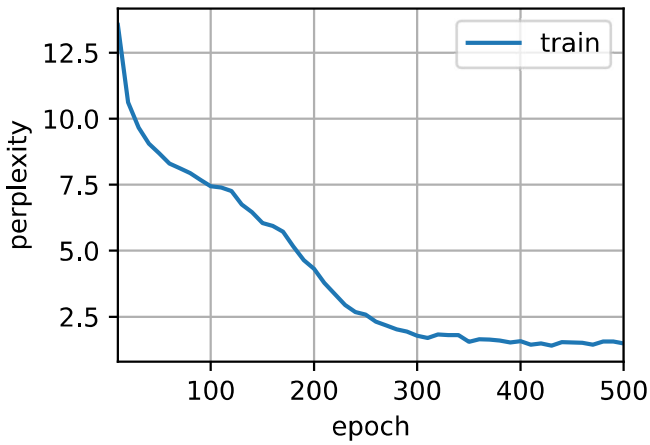
```
perplexity 1.0, 69604.6 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby
```




Finally, let us check the results of using the random sampling method.

```
net = RNNModelScratch(len(vocab), num_hiddens, d2l.try_gpu(), get_params,
                      init_rnn_state, rnn)
train_ch8(net, train_iter, vocab, lr, num_epochs, d2l.try_gpu(),
          use_random_iter=True)
```

```
perplexity 1.5, 68624.1 tokens/sec on cuda:0
time travellerit s against reason said filbywhat i umuntable thi
travellerit s against reason said filbywhat i umuntable thi
```




While implementing the above RNN model from scratch is instructive, it is not convenient. In the next section we will see how to improve the RNN model, such as how to make it easier to implement and make it run faster.

## Summary

- We can train an RNN-based character-level language model to generate text following the user-provided text prefix.
- A simple RNN language model consists of input encoding, RNN modeling, and output generation.
- RNN models need state initialization for training, though random sampling and sequential partitioning use different ways.
- When using sequential partitioning, we need to detach the gradient to reduce computational cost.
- A warm-up period allows a model to update itself (e.g., obtain a better hidden state than its initialized value) before making any prediction.
- Gradient clipping prevents gradient explosion, but it cannot fix vanishing gradients.

### Exercises

1. Show that one-hot encoding is equivalent to picking a different embedding for each object.
2. Adjust the hyperparameters (e.g., number of epochs, number of hidden units, number of time steps in a minibatch, and learning rate) to improve the perplexity.
   - How low can you go?
   - Replace one-hot encoding with learnable embeddings. Does this lead to better performance?
   - How well will it work on other books by H. G. Wells, e.g., *The War of the Worlds*[106]?
3. Modify the prediction function such as to use sampling rather than picking the most likely next character.
   - What happens?
   - Bias the model towards more likely outputs, e.g., by sampling from $q(x_t \mid x_{t-1}, \ldots, x_1) \propto P(x_t \mid x_{t-1}, \ldots, x_1)^\alpha$ for $\alpha > 1$.
4. Run the code in this section without clipping the gradient. What happens?
5. Change sequential partitioning so that it does not separate hidden states from the computational graph. Does the running time change? How about the perplexity?
6. Replace the activation function used in this section with ReLU and repeat the experiments in this section. Do we still need gradient clipping? Why?

## 8.6 Concise Implementation of Recurrent Neural Networks

While Section 8.5 was instructive to see how RNNs are implemented, this is not convenient or fast. This section will show how to implement the same language model more efficiently using functions provided by high-level APIs of a deep learning framework. We begin as before by reading the time machine dataset.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

### 8.6.1 Defining the Model

High-level APIs provide implementations of recurrent neural networks. We construct the recurrent neural network layer `rnn_layer` with a single hidden layer and 256 hidden units. In fact, we have not even discussed yet what it means to have multiple layers—this will happen in Section 9.3. For now, suffice it to say that multiple layers simply amount to the output of one layer of RNN being used as the input for the next layer of RNN.

```
num_hiddens = 256
rnn_layer = nn.RNN(len(vocab), num_hiddens)
```

We use a tensor to initialize the hidden state, whose shape is (number of hidden layers, batch size, number of hidden units).

```
state = torch.zeros((1, batch_size, num_hiddens))
state.shape
```

```
torch.Size([1, 32, 256])
```

With a hidden state and an input, we can compute the output with the updated hidden state. It should be emphasized that the "output" (`Y`) of `rnn_layer` does *not* involve computation of output layers: it refers to the hidden state at *each* time step, and they can be used as the input to the subsequent output layer.

```
X = torch.rand(size=(num_steps, batch_size, len(vocab)))
Y, state_new = rnn_layer(X, state)
Y.shape, state_new.shape
```

```
(torch.Size([35, 32, 256]), torch.Size([1, 32, 256]))
```

Similar to Section 8.5, we define an `RNNModel` class for a complete RNN model. Note that `rnn_layer` only contains the hidden recurrent layers, we need to create a separate output layer.

```
#@save
class RNNModel(nn.Module):
    """The RNN model."""
    def __init__(self, rnn_layer, vocab_size, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        self.rnn = rnn_layer
        self.vocab_size = vocab_size
        self.num_hiddens = self.rnn.hidden_size
        # If the RNN is bidirectional (to be introduced later),
        # `num_directions` should be 2, else it should be 1.
        if not self.rnn.bidirectional:
            self.num_directions = 1
            self.linear = nn.Linear(self.num_hiddens, self.vocab_size)
        else:
            self.num_directions = 2
            self.linear = nn.Linear(self.num_hiddens * 2, self.vocab_size)

    def forward(self, inputs, state):
        X = F.one_hot(inputs.T.long(), self.vocab_size)
```

(continues on next page)

(continued from previous page)

```
        X = X.to(torch.float32)
        Y, state = self.rnn(X, state)
        # The fully connected layer will first change the shape of `Y` to
        # (`num_steps` * `batch_size`, `num_hiddens`). Its output shape is
        # (`num_steps` * `batch_size`, `vocab_size`).
        output = self.linear(Y.reshape((-1, Y.shape[-1])))
        return output, state

    def begin_state(self, device, batch_size=1):
        if not isinstance(self.rnn, nn.LSTM):
            # `nn.GRU` takes a tensor as hidden state
            return  torch.zeros((self.num_directions * self.rnn.num_layers,
                                 batch_size, self.num_hiddens),
                                device=device)
        else:
            # `nn.LSTM` takes a tuple of hidden states
            return (torch.zeros((
                self.num_directions * self.rnn.num_layers,
                batch_size, self.num_hiddens), device=device),
                    torch.zeros((
                        self.num_directions * self.rnn.num_layers,
                        batch_size, self.num_hiddens), device=device))
```

### 8.6.2 Training and Predicting

Before training the model, let us make a prediction with the a model that has random weights.

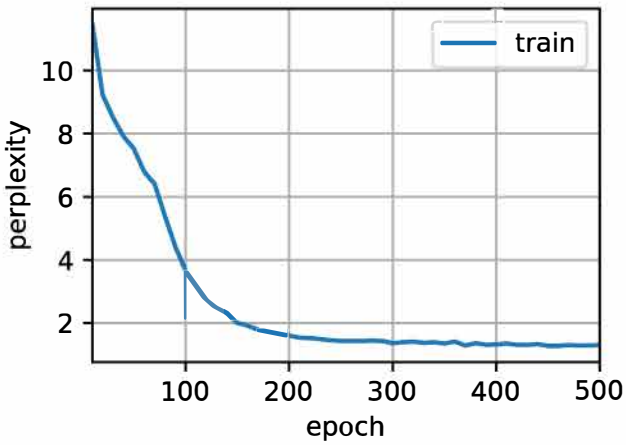
```
device = d2l.try_gpu()
net = RNNModel(rnn_layer, vocab_size=len(vocab))
net = net.to(device)
d2l.predict_ch8('time traveller', 10, net, vocab, device)
```

```
'time travellerfrf pff f '
```

As is quite obvious, this model does not work at all. Next, we call `train_ch8` with the same hyperparameters defined in Section 8.5 and train our model with high-level APIs.

```
num_epochs, lr = 500, 1
d2l.train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.3, 298788.1 tokens/sec on cuda:0
time traveller held in his hand was a glitteringmetans lived dit
traveller afoer a wommertale mave at boub in the foree me b
```




Compared with the last section, this model achieves comparable perplexity, albeit within a shorter period of time, due to the code being more optimized by high-level APIs of the deep learning framework.

## Summary

- High-level APIs of the deep learning framework provides an implementation of the RNN layer.
- The RNN layer of high-level APIs returns an output and an updated hidden state, where the output does not involve output layer computation.
- Using high-level APIs leads to faster RNN training than using its implementation from scratch.

## Exercises

1. Can you make the RNN model overfit using the high-level APIs?
2. What happens if you increase the number of hidden layers in the RNN model? Can you make the model work?
3. Implement the autoregressive model of Section 8.1 using an RNN.

## 8.7 Backpropagation Through Time

So far we have repeatedly alluded to things like *exploding gradients*, *vanishing gradients*, and the need to *detach the gradient* for RNNs. For instance, in Section 8.5 we invoked the `detach` function on the sequence. None of this was really fully explained, in the interest of being able to build a model quickly and to see how it works. In this section, we will delve a bit more deeply into the details of backpropagation for sequence models and why (and how) the mathematics works.

We encountered some of the effects of gradient explosion when we first implemented RNNs (Section 8.5). In particular, if you solved the exercises, you would have seen that gradient clipping is vital to ensure proper convergence. To provide a better understanding of this issue, this section will review how gradients are computed for sequence models. Note that there is nothing conceptually new in how it works. After all, we are still merely applying the chain rule to compute gradients. Nonetheless, it is worth while reviewing backpropagation (Section 4.7) again.

We have described forward and backward propagations and computational graphs in MLPs in Section 4.7. Forward propagation in an RNN is relatively straightforward. *Backpropagation through time* is actually a specific application of backpropagation in RNNs (Werbos, 1990). It requires us to expand the computational graph of an RNN one time step at a time to obtain the dependencies among model variables and parameters. Then, based on the chain rule, we apply backpropagation to compute and store gradients. Since sequences can be rather long, the dependency can be rather lengthy. For instance, for a sequence of 1000 characters, the first token could potentially have significant influence on the token at the final position. This is not really computationally feasible (it takes too long and requires too much memory) and it requires over 1000 matrix products before we would arrive at that very elusive gradient. This is a process fraught with computational and statistical uncertainty. In the following we will elucidate what happens and how to address this in practice.

### 8.7.1 Analysis of Gradients in RNNs

We start with a simplified model of how an RNN works. This model ignores details about the specifics of the hidden state and how it is updated. The mathematical notation here does not explicitly distinguish scalars, vectors, and matrices as it used to do. These details are immaterial to the analysis and would only serve to clutter the notation in this subsection.

In this simplified model, we denote $h_t$ as the hidden state, $x_t$ as the input, and $o_t$ as the output at time step $t$. Recall our discussions in Section 8.4.2 that the input and the hidden state can be concatenated to be multiplied by one weight variable in the hidden layer. Thus, we use $w_h$ and $w_o$ to indicate the weights of the hidden layer and the output layer, respectively. As a result, the hidden states and outputs at each time steps can be explained as

$$\begin{aligned} h_t &= f(x_t, h_{t-1}, w_h), \\ o_t &= g(h_t, w_o), \end{aligned} \tag{8.7.1}$$

where $f$ and $g$ are transformations of the hidden layer and the output layer, respectively. Hence, we have a chain of values $\{\ldots, (x_{t-1}, h_{t-1}, o_{t-1}), (x_t, h_t, o_t), \ldots\}$ that depend on each other via recurrent computation. The forward propagation is fairly straightforward. All we need is to loop through the $(x_t, h_t, o_t)$ triples one time step at a time. The discrepancy between output $o_t$ and the desired label $y_t$ is then evaluated by an objective function across all the $T$ time steps as

$$L(x_1, \ldots, x_T, y_1, \ldots, y_T, w_h, w_o) = \frac{1}{T} \sum_{t=1}^{T} l(y_t, o_t). \tag{8.7.2}$$

For backpropagation, matters are a bit trickier, especially when we compute the gradients with regard to the parameters $w_h$ of the objective function $L$. To be specific, by the chain rule,

$$\begin{aligned}\frac{\partial L}{\partial w_h} &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t, o_t)}{\partial w_h} \\ &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t, o_t)}{\partial o_t}\frac{\partial g(h_t, w_o)}{\partial h_t}\frac{\partial h_t}{\partial w_h}.\end{aligned} \tag{8.7.3}$$

The first and the second factors of the product in (8.7.3) are easy to compute. The third factor $\partial h_t/\partial w_h$ is where things get tricky, since we need to recurrently compute the effect of the parameter $w_h$ on $h_t$. According to the recurrent computation in (8.7.1), $h_t$ depends on both $h_{t-1}$ and $w_h$, where computation of $h_{t-1}$ also depends on $w_h$. Thus, using the chain rule yields

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial w_h}. \tag{8.7.4}$$

To derive the above gradient, assume that we have three sequences $\{a_t\}, \{b_t\}, \{c_t\}$ satisfying $a_0 = 0$ and $a_t = b_t + c_t a_{t-1}$ for $t = 1, 2, \ldots$. Then for $t \geq 1$, it is easy to show

$$a_t = b_t + \sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t} c_j\right) b_i. \tag{8.7.5}$$

By substituting $a_t$, $b_t$, and $c_t$ according to

$$\begin{aligned}a_t &= \frac{\partial h_t}{\partial w_h}, \\ b_t &= \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h}, \\ c_t &= \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}},\end{aligned} \tag{8.7.6}$$

the gradient computation in (8.7.4) satisfies $a_t = b_t + c_t a_{t-1}$. Thus, per (8.7.5), we can remove the recurrent computation in (8.7.4) with

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t}\frac{\partial f(x_j, h_{j-1}, w_h)}{\partial h_{j-1}}\right)\frac{\partial f(x_i, h_{i-1}, w_h)}{\partial w_h}. \tag{8.7.7}$$

While we can use the chain rule to compute $\partial h_t/\partial w_h$ recursively, this chain can get very long whenever $t$ is large. Let us discuss a number of strategies for dealing with this problem.

### Full Computation

Obviously, we can just compute the full sum in (8.7.7). However, this is very slow and gradients can blow up, since subtle changes in the initial conditions can potentially affect the outcome a lot. That is, we could see things similar to the butterfly effect where minimal changes in the initial conditions lead to disproportionate changes in the outcome. This is actually quite undesirable in terms of the model that we want to estimate. After all, we are looking for robust estimators that generalize well. Hence this strategy is almost never used in practice.

### Truncating Time Steps

Alternatively, we can truncate the sum in (8.7.7) after $\tau$ steps. This is what we have been discussing so far, such as when we detached the gradients in Section 8.5. This leads to an *approximation* of the true gradient, simply by terminating the sum at $\partial h_{t-\tau}/\partial w_h$. In practice this works quite well. It is what is commonly referred to as truncated backpropgation through time (Jaeger, 2002). One of the consequences of this is that the model focuses primarily on short-term influence rather than long-term consequences. This is actually *desirable*, since it biases the estimate towards simpler and more stable models.

### Randomized Truncation

Last, we can replace $\partial h_t/\partial w_h$ by a random variable which is correct in expectation but truncates the sequence. This is achieved by using a sequence of $\xi_t$ with predefined $0 \leq \pi_t \leq 1$, where $P(\xi_t = 0) = 1 - \pi_t$ and $P(\xi_t = \pi_t^{-1}) = \pi_t$, thus $E[\xi_t] = 1$. We use this to replace the gradient $\partial h_t/\partial w_h$ in (8.7.4) with

$$z_t = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \xi_t \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_h}. \tag{8.7.8}$$

It follows from the definition of $\xi_t$ that $E[z_t] = \partial h_t/\partial w_h$. Whenever $\xi_t = 0$ the recurrent computation terminates at that time step $t$. This leads to a weighted sum of sequences of varying lengths where long sequences are rare but appropriately overweighted. This idea was proposed by Tallec and Ollivier (Tallec & Ollivier, 2017).

### Comparing Strategies

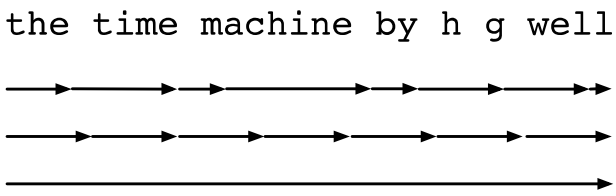



Fig. 8.7.1: Comparing strategies for computing gradients in RNNs. From top to bottom: randomized truncation, regular truncation, and full computation.

Fig. 8.7.1 illustrates the three strategies when analyzing the first few characters of *The Time Machine* book using backpropagation through time for RNNs:

- The first row is the randomized truncation that partitions the text into segments of varying lengths.
- The second row is the regular truncation that breaks the text into subsequences of the same length. This is what we have been doing in RNN experiments.
- The third row is the full backpropagation through time that leads to a computationally infeasible expression.

Unfortunately, while appealing in theory, randomized truncation does not work much better than regular truncation, most likely due to a number of factors. First, the effect of an observation after

a number of backpropagation steps into the past is quite sufficient to capture dependencies in practice. Second, the increased variance counteracts the fact that the gradient is more accurate with more steps. Third, we actually *want* models that have only a short range of interactions. Hence, regularly truncated backpropagation through time has a slight regularizing effect that can be desirable.

### 8.7.2 Backpropagation Through Time in Detail

After discussing the general principle, let us discuss backpropagation through time in detail. Different from the analysis in Section 8.7.1, in the following we will show how to compute the gradients of the objective function with respect to all the decomposed model parameters. To keep things simple, we consider an RNN without bias parameters, whose activation function in the hidden layer uses the identity mapping ($\phi(x) = x$). For time step $t$, let the single example input and the label be $\mathbf{x}_t \in \mathbb{R}^d$ and $y_t$, respectively. The hidden state $\mathbf{h}_t \in \mathbb{R}^h$ and the output $\mathbf{o}_t \in \mathbb{R}^q$ are computed as

$$\begin{aligned}\mathbf{h}_t &= \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1},\\ \mathbf{o}_t &= \mathbf{W}_{qh}\mathbf{h}_t,\end{aligned} \tag{8.7.9}$$

where $\mathbf{W}_{hx} \in \mathbb{R}^{h\times d}$, $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$, and $\mathbf{W}_{qh} \in \mathbb{R}^{q\times h}$ are the weight parameters. Denote by $l(\mathbf{o}_t, y_t)$ the loss at time step $t$. Our objective function, the loss over $T$ time steps from the beginning of the sequence is thus

$$L = \frac{1}{T}\sum_{t=1}^{T} l(\mathbf{o}_t, y_t). \tag{8.7.10}$$

In order to visualize the dependencies among model variables and parameters during computation of the RNN, we can draw a computational graph for the model, as shown in Fig. 8.7.2. For example, the computation of the hidden states of time step 3, $\mathbf{h}_3$, depends on the model parameters $\mathbf{W}_{hx}$ and $\mathbf{W}_{hh}$, the hidden state of the last time step $\mathbf{h}_2$, and the input of the current time step $\mathbf{x}_3$.

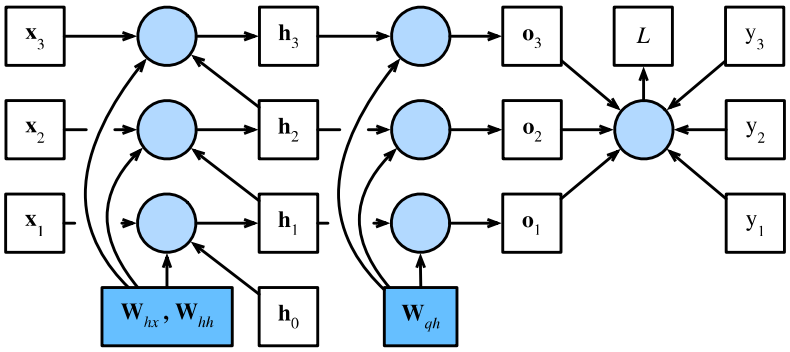



Fig. 8.7.2: Computational graph showing dependencies for an RNN model with three time steps. Boxes represent variables (not shaded) or parameters (shaded) and circles represent operators.

As just mentioned, the model parameters in Fig. 8.7.2 are $\mathbf{W}_{hx}$, $\mathbf{W}_{hh}$, and $\mathbf{W}_{qh}$. Generally, training this model requires gradient computation with respect to these parameters $\partial L/\partial \mathbf{W}_{hx}$, $\partial L/\partial \mathbf{W}_{hh}$, and $\partial L/\partial \mathbf{W}_{qh}$. According to the dependencies in Fig. 8.7.2, we can traverse in the opposite direction of the arrows to calculate and store the gradients in turn. To flexibly express the multiplication of matrices, vectors, and scalars of different shapes in the chain rule, we continue to use the prod operator as described in Section 4.7.

First of all, differentiating the objective function with respect to the model output at any time step $t$ is fairly straightforward:

$$\frac{\partial L}{\partial \mathbf{o}_t} = \frac{\partial l(\mathbf{o}_t, y_t)}{T \cdot \partial \mathbf{o}_t} \in \mathbb{R}^q. \tag{8.7.11}$$

Now, we can calculate the gradient of the objective function with respect to the parameter $\mathbf{W}_{qh}$ in the output layer: $\partial L / \partial \mathbf{W}_{qh} \in \mathbb{R}^{q \times h}$. Based on Fig. 8.7.2, the objective function $L$ depends on $\mathbf{W}_{qh}$ via $\mathbf{o}_1, \ldots, \mathbf{o}_T$. Using the chain rule yields

$$\frac{\partial L}{\partial \mathbf{W}_{qh}} = \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{qh}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{o}_t} \mathbf{h}_t^\top, \tag{8.7.12}$$

where $\partial L / \partial \mathbf{o}_t$ is given by (8.7.11).

Next, as shown in Fig. 8.7.2, at the final time step $T$ the objective function $L$ depends on the hidden state $\mathbf{h}_T$ only via $\mathbf{o}_T$. Therefore, we can easily find the gradient $\partial L / \partial \mathbf{h}_T \in \mathbb{R}^h$ using the chain rule:

$$\frac{\partial L}{\partial \mathbf{h}_T} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_T}, \frac{\partial \mathbf{o}_T}{\partial \mathbf{h}_T}\right) = \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_T}. \tag{8.7.13}$$

It gets trickier for any time step $t < T$, where the objective function $L$ depends on $\mathbf{h}_t$ via $\mathbf{h}_{t+1}$ and $\mathbf{o}_t$. According to the chain rule, the gradient of the hidden state $\partial L / \partial \mathbf{h}_t \in \mathbb{R}^h$ at any time step $t < T$ can be recurrently computed as:

$$\frac{\partial L}{\partial \mathbf{h}_t} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_{t+1}}, \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}\right) + \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\right) = \mathbf{W}_{hh}^\top \frac{\partial L}{\partial \mathbf{h}_{t+1}} + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_t}. \tag{8.7.14}$$

For analysis, expanding the recurrent computation for any time step $1 \leq t \leq T$ gives

$$\frac{\partial L}{\partial \mathbf{h}_t} = \sum_{i=t}^{T} \left(\mathbf{W}_{hh}^\top\right)^{T-i} \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{T+t-i}}. \tag{8.7.15}$$

We can see from (8.7.15) that this simple linear example already exhibits some key problems of long sequence models: it involves potentially very large powers of $\mathbf{W}_{hh}^\top$. In it, eigenvalues smaller than 1 vanish and eigenvalues larger than 1 diverge. This is numerically unstable, which manifests itself in the form of vanishing and exploding gradients. One way to address this is to truncate the time steps at a computationally convenient size as discussed in Section 8.7.1. In practice, this truncation is effected by detaching the gradient after a given number of time steps. Later on we will see how more sophisticated sequence models such as long short-term memory can alleviate this further.

Finally, Fig. 8.7.2 shows that the objective function $L$ depends on model parameters $\mathbf{W}_{hx}$ and $\mathbf{W}_{hh}$ in the hidden layer via hidden states $\mathbf{h}_1, \ldots, \mathbf{h}_T$. To compute gradients with respect to such parameters $\partial L / \partial \mathbf{W}_{hx} \in \mathbb{R}^{h \times d}$ and $\partial L / \partial \mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$, we apply the chain rule that gives

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{W}_{hx}} &= \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hx}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{x}_t^\top, \\
\frac{\partial L}{\partial \mathbf{W}_{hh}} &= \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{h}_{t-1}^\top,
\end{aligned} \tag{8.7.16}$$

where $\partial L / \partial \mathbf{h}_t$ that is recurrently computed by (8.7.13) and (8.7.14) is the key quantity that affects the numerical stability.

Since backpropagation through time is the application of backpropagation in RNNs, as we have explained in Section 4.7, training RNNs alternates forward propagation with backpropagation through time. Besides, backpropagation through time computes and stores the above gradients in turn. Specifically, stored intermediate values are reused to avoid duplicate calculations, such as storing $\partial L/\partial \mathbf{h}_t$ to be used in computation of both $\partial L/\partial \mathbf{W}_{hx}$ and $\partial L/\partial \mathbf{W}_{hh}$.

## Summary

- Backpropagation through time is merely an application of backpropagation to sequence models with a hidden state.
- Truncation is needed for computational convenience and numerical stability, such as regular truncation and randomized truncation.
- High powers of matrices can lead to divergent or vanishing eigenvalues. This manifests itself in the form of exploding or vanishing gradients.
- For efficient computation, intermediate values are cached during backpropagation through time.

## Exercises

1. Assume that we have a symmetric matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$ with eigenvalues $\lambda_i$ whose corresponding eigenvectors are $\mathbf{v}_i$ ($i = 1, \ldots, n$). Without loss of generality, assume that they are ordered in the order $|\lambda_i| \geq |\lambda_{i+1}|$.
2. Show that $\mathbf{M}^k$ has eigenvalues $\lambda_i^k$.
3. Prove that for a random vector $\mathbf{x} \in \mathbb{R}^n$, with high probability $\mathbf{M}^k\mathbf{x}$ will be very much aligned with the eigenvector $\mathbf{v}_1$ of $\mathbf{M}$. Formalize this statement.
4. What does the above result mean for gradients in RNNs?
5. Besides gradient clipping, can you think of any other methods to cope with gradient explosion in recurrent neural networks?