14 | Natural Language Processing

Natural language processing is concerned with interactions between computers and humans that use natural language. In practice, it is very common for us to use this technique to process and analyze large amounts of natural language data, like the language models from Chapter 8

In this chapter, we will discuss how to use vectors to represent words and train the word vectors on a corpus. We will also use word vectors pre-trained on a larger corpus to find synonyms and analogies. Then, in the text classification task, we will use word vectors to analyze the emotion of a text and explain the important ideas of timing data classification based on recurrent neural networks and the convolutional neural networks.

14.1 Word Embedding (word2vec)

A natural language is a complex system that we use to express meanings. In this system, words are the basic unit of linguistic meaning. As its name implies, a word vector is a vector used to represent a word. It can also be thought of as the feature vector of a word. The technique of mapping words to vectors of real numbers is also known as word embedding. Over the last few years, word embedding has gradually become basic knowledge in natural language processing.

14.1.1 Why Not Use One-hot Vectors?

We used one-hot vectors to represent words (characters are words) in Section 8.5 . Recall that when we assume the number of different words in a dictionary (the dictionary size) is N, each word can correspond one-to-one with consecutive integers from 0 to $N-1$. These integers that correspond to words are called the indices of the words. We assume that the index of a word is i. In order to get the one-hot vector representation of the word, we create a vector of all 0s with a length of N and set element i to 1. In this way, each word is represented as a vector of length N that can be used directly by the neural network.

Although one-hot word vectors are easy to construct, they are usually not a good choice. One of the major reasons is that the one-hot word vectors cannot accurately express the similarity between different words, such as the cosine similarity that we commonly use. For the vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their cosine similarities are the cosines of the angles between them:

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \in [-1, 1]. \tag{14.1.1}$$

Since the cosine similarity between the one-hot vectors of any two different words is 0, it is difficult to use the one-hot vector to accurately represent the similarity between multiple different words.

Word2vec[216] is a tool that we came up with to solve the problem above. It represents each word with a fixed-length vector and uses these vectors to better indicate the similarity and analogy relationships between different words. The Word2vec tool contains two models: skip-gram (Mikolov et al., 2013b) and continuous bag of words (CBOW) (Mikolov et al., 2013a). Next, we will take a look at the two models and their training methods.

14.1.2 The Skip-Gram Model

The skip-gram model assumes that a word can be used to generate the words that surround it in a text sequence. For example, we assume that the text sequence is "the", "man", "loves", "his", and "son". We use "loves" as the central target word and set the context window size to 2. As shown in Fig. 14.1.1, given the central target word "loves", the skip-gram model is concerned with the conditional probability for generating the context words, "the", "man", "his" and "son", that are within a distance of no more than 2 words, which is

$$P(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} \mid \text{"loves"}). \tag{14.1.2}$$

We assume that, given the central target word, the context words are generated independently of each other. In this case, the formula above can be rewritten as

$$P(\text{"the"} \mid \text{"loves"}) \cdot P(\text{"man"} \mid \text{"loves"}) \cdot P(\text{"his"} \mid \text{"loves"}) \cdot P(\text{"son"} \mid \text{"loves"}). \tag{14.1.3}$$

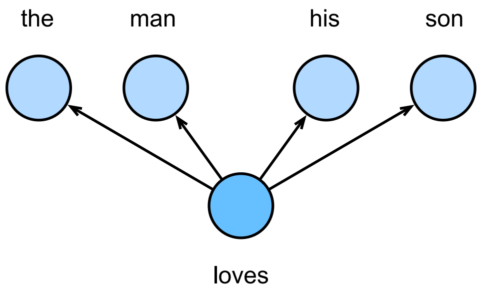


Fig. 14.1.1: The skip-gram model cares about the conditional probability of generating context words for a given central target word.

In the skip-gram model, each word is represented as two d-dimension vectors, which are used to compute the conditional probability. We assume that the word is indexed as i in the dictionary, its vector is represented as $\mathbf{v}_i \in \mathbb{R}^d$ when it is the central target word, and $\mathbf{u}_i \in \mathbb{R}^d$ when it is a context word. Let the central target word w_c and context word w_o be indexed as c and o respectively in the dictionary. The conditional probability of generating the context word for the given central target word can be obtained by performing a softmax operation on the vector inner product:

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}, \tag{14.1.4}$$

where vocabulary index set $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}|-1\}$. Assume that a text sequence of length T is given, where the word at timestep t is denoted as $w^{(t)}$. Assume that context words are independently

[216] https://code.google.com/archive/p/word2vec/

generated given center words. When context window size is m, the likelihood function of the skip-gram model is the joint probability of generating all the context words given any center word

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(w^{(t+j)} \mid w^{(t)}), \tag{14.1.5}$$

Here, any timestep that is less than 1 or greater than T can be ignored.

Skip-Gram Model Training

The skip-gram model parameters are the central target word vector and context word vector for each individual word. In the training process, we are going to learn the model parameters by maximizing the likelihood function, which is also known as maximum likelihood estimation. This is equivalent to minimizing the following loss function:

$$-\sum_{t=1}^{T} \sum_{-m \leq j \leq m,\ j \neq 0} \log P(w^{(t+j)} \mid w^{(t)}). \tag{14.1.6}$$

If we use the SGD, in each iteration we are going to pick a shorter subsequence through random sampling to compute the loss for that subsequence, and then compute the gradient to update the model parameters. The key of gradient computation is to compute the gradient of the logarithmic conditional probability for the central word vector and the context word vector. By definition, we first have

$$\log P(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log\left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)\right). \tag{14.1.7}$$

Through differentiation, we can get the gradient $\mathbf{v}_c$ from the formula above.

$$\begin{aligned}
\frac{\partial \log P(w_o \mid w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c)\mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \\
&= \mathbf{u}_o - \sum_{j \in \mathcal{V}} \left(\frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}\right) \mathbf{u}_j \\
&= \mathbf{u}_o - \sum_{j \in \mathcal{V}} P(w_j \mid w_c)\mathbf{u}_j.
\end{aligned} \tag{14.1.8}$$

Its computation obtains the conditional probability for all the words in the dictionary given the central target word w_c. We then use the same method to obtain the gradients for other word vectors.

After the training, for any word in the dictionary with index i, we are going to get its two word vector sets $\mathbf{v}_i$ and $\mathbf{u}_i$. In applications of natural language processing (NLP), the central target word vector in the skip-gram model is generally used as the representation vector of a word.

14.1.3 The Continuous Bag of Words (CBOW) Model

The continuous bag of words (CBOW) model is similar to the skip-gram model. The biggest difference is that the CBOW model assumes that the central target word is generated based on the context words before and after it in the text sequence. With the same text sequence "the", "man", "loves", "his" and "son", in which "loves" is the central target word, given a context window size of 2, the CBOW model is concerned with the conditional probability of generating the target word "loves" based on the context words "the", "man", "his" and "son"(as shown in Fig. 14.1.2), such as

$$P(\text{"loves"} \mid \text{"the"}, \text{"man"}, \text{"his"}, \text{"son"}). \tag{14.1.9}$$

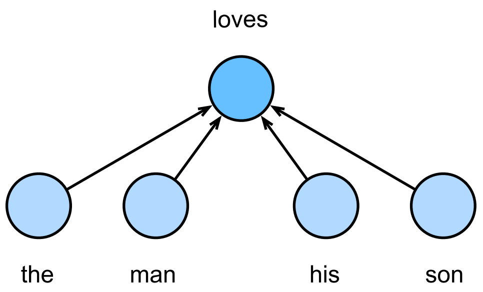


Fig. 14.1.2: The CBOW model cares about the conditional probability of generating the central target word from given context words.

Since there are multiple context words in the CBOW model, we will average their word vectors and then use the same method as the skip-gram model to compute the conditional probability. We assume that $\mathbf{v_i} \in \mathbb{R}^d$ and $\mathbf{u_i} \in \mathbb{R}^d$ are the context word vector and central target word vector of the word with index i in the dictionary (notice that the symbols are opposite to the ones in the skip-gram model). Let central target word w_c be indexed as c, and context words $w_{o_1}, \ldots, w_{o_{2m}}$ be indexed as $o_1, \ldots, o_{2m}$ in the dictionary. Thus, the conditional probability of generating a central target word from the given context word is

$$P(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\mathbf{u}_c^\top (\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\mathbf{u}_i^\top (\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}. \tag{14.1.10}$$

For brevity, denote $\mathcal{W}_o = \{w_{o_1}, \ldots, w_{o_{2m}}\}$, and $\bar{\mathbf{v}}_o = (\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})/(2m)$. The equation above can be simplified as

$$P(w_c \mid \mathcal{W}_o) = \frac{\exp\left(\mathbf{u}_c^\top \bar{\mathbf{v}}_o\right)}{\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right)}. \tag{14.1.11}$$

Given a text sequence of length T, we assume that the word at timestep t is $w^{(t)}$, and the context window size is m. The likelihood function of the CBOW model is the probability of generating any central target word from the context words.

$$\prod_{t=1}^{T} P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{14.1.12}$$

CBOW Model Training

CBOW model training is quite similar to skip-gram model training. The maximum likelihood estimation of the CBOW model is equivalent to minimizing the loss function.

$$-\sum_{t=1}^{T} \log P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{14.1.13}$$

Notice that

$$\log\, P(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left(\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o \right) \right). \tag{14.1.14}$$

Through differentiation, we can compute the logarithm of the conditional probability of the gradient of any context word vector $\mathbf{v}_{o_i}(i = 1, \ldots, 2m)$ in the formula above.

$$\frac{\partial \log\, P(w_c \mid \mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \bar{\mathbf{v}}_o) \mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)} \right) = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} P(w_j \mid \mathcal{W}_o) \mathbf{u}_j \right). \tag{14.1.15}$$

We then use the same method to obtain the gradients for other word vectors. Unlike the skip-gram model, we usually use the context word vector as the representation vector for a word in the CBOW model.

Summary

- A word vector is a vector used to represent a word. The technique of mapping words to vectors of real numbers is also known as word embedding.
- Word2vec includes both the continuous bag of words (CBOW) and skip-gram models. The skip-gram model assumes that context words are generated based on the central target word. The CBOW model assumes that the central target word is generated based on the context words.

Exercises

1. What is the computational complexity of each gradient? If the dictionary contains a large volume of words, what problems will this cause?
2. There are some fixed phrases in the English language which consist of multiple words, such as "new york". How can you train their word vectors? Hint: See section 4 in the Word2vec paper[2].
3. Use the skip-gram model as an example to think about the design of a word2vec model. What is the relationship between the inner product of two word vectors and the cosine similarity in the skip-gram model? For a pair of words with close semantical meaning, why it is likely for their word vector cosine similarity to be high?

14.2 Approximate Training for Word2vec

Recall content of the last section. The core feature of the skip-gram model is the use of softmax operations to compute the conditional probability of generating context word w_o based on the given central target word w_c.

$$P(w_o \mid w_c) = \frac{\text{exp}(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \text{exp}(\mathbf{u}_i^\top \mathbf{v}_c)}. \tag{14.2.1}$$

The logarithmic loss corresponding to the conditional probability is given as

$$-\log P(w_o \mid w_c) = -\mathbf{u}_o^\top \mathbf{v}_c + \log\left(\sum_{i \in \mathcal{V}} \text{exp}(\mathbf{u}_i^\top \mathbf{v}_c)\right). \tag{14.2.2}$$

Because the softmax operation has considered that the context word could be any word in the dictionary $\mathcal{V}$, the loss mentioned above actually includes the sum of the number of items in the dictionary size. From the last section, we know that for both the skip-gram model and CBOW model, because they both get the conditional probability using a softmax operation, the gradient computation for each step contains the sum of the number of items in the dictionary size. For larger dictionaries with hundreds of thousands or even millions of words, the overhead for computing each gradient may be too high. In order to reduce such computational complexity, we will introduce two approximate training methods in this section: negative sampling and hierarchical softmax. Since there is no major difference between the skip-gram model and the CBOW model, we will only use the skip-gram model as an example to introduce these two training methods in this section.

14.2.1 Negative Sampling

Negative sampling modifies the original objective function. Given a context window for the central target word w_c, we will treat it as an event for context word w_o to appear in the context window and compute the probability of this event from

$$P(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c), \tag{14.2.3}$$

Here, the σ function has the same definition as the sigmoid activation function:

$$\sigma(x) = \frac{1}{1 + \text{exp}(-x)}. \tag{14.2.4}$$

We will first consider training the word vector by maximizing the joint probability of all events in the text sequence. Given a text sequence of length T, we assume that the word at timestep t is $w^{(t)}$ and the context window size is m. Now we consider maximizing the joint probability

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(D = 1 \mid w^{(t)}, w^{(t+j)}). \tag{14.2.5}$$

However, the events included in the model only consider positive examples. In this case, only when all the word vectors are equal and their values approach infinity can the joint probability above be maximized to 1. Obviously, such word vectors are meaningless. Negative sampling makes the objective function more meaningful by sampling with an addition of negative examples. Assume that event P occurs when context word w_o appears in the context window of central target word w_c, and we sample K words that do not appear in the context window according to the distribution $P(w)$ to act as noise words. We assume the event for noise word $w_k (k = 1, \ldots, K)$ to not appear in the context window of central target word w_c is N_k. Suppose that events P and $N_1, \ldots, N_K$ for both positive and negative examples are independent of each other. By considering negative sampling, we can rewrite the joint probability above, which only considers the positive examples, as

$$\prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \ne 0} P(w^{(t+j)} \mid w^{(t)}), \tag{14.2.6}$$

Here, the conditional probability is approximated to be

$$P(w^{(t+j)} \mid w^{(t)}) = P(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1,\ w_k \sim P(w)}^{K} P(D = 0 \mid w^{(t)}, w_k). \tag{14.2.7}$$

Let the text sequence index of word $w^{(t)}$ at timestep t be i_t and h_k for noise word w_k in the dictionary. The logarithmic loss for the conditional probability above is

$$\begin{aligned} -\log P(w^{(t+j)} \mid w^{(t)}) =& -\log P(D = 1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log P(D = 0 \mid w^{(t)}, w_k) \\ =& -\log \sigma\left(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log\left(1 - \sigma\left(\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}\right)\right) \\ =& -\log \sigma\left(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log \sigma\left(-\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}\right). \end{aligned} \tag{14.2.8}$$

Here, the gradient computation in each step of the training is no longer related to the dictionary size, but linearly related to K. When K takes a smaller constant, the negative sampling has a lower computational overhead for each step.

14.2.2 Hierarchical Softmax

Hierarchical softmax is another type of approximate training method. It uses a binary tree for data structure as illustrated in Fig. 14.2.1, with the leaf nodes of the tree representing every word in the dictionary $\mathcal{V}$.

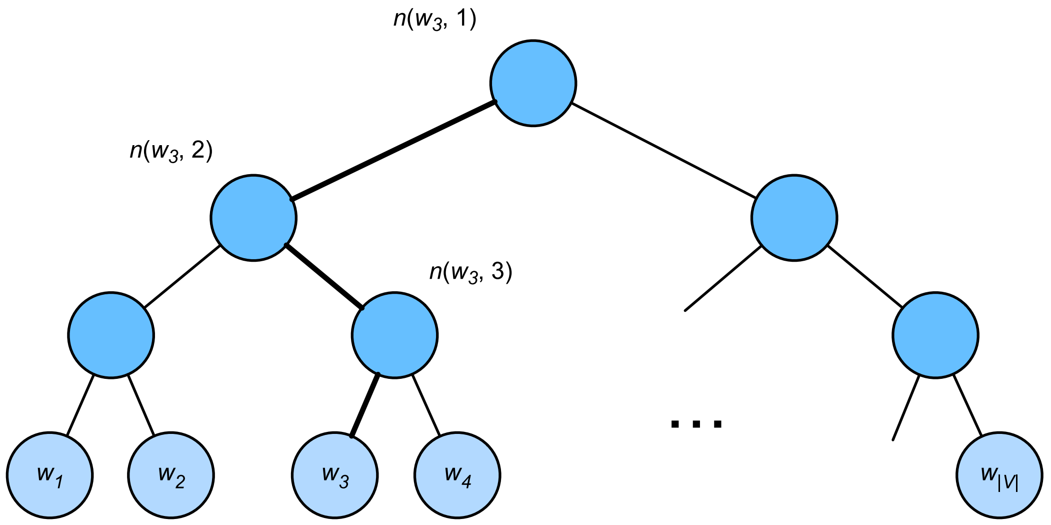


Fig. 14.2.1: Hierarchical Softmax. Each leaf node of the tree represents a word in the dictionary.

We assume that $L(w)$ is the number of nodes on the path (including the root and leaf nodes) from the root node of the binary tree to the leaf node of word w. Let $n(w, j)$ be the j^{th} node on this path, with the context word vector $\mathbf{u}_{n(w,j)}$. We use Figure 10.3 as an example, so $L(w_3) = 4$. Hierarchical softmax will approximate the conditional probability in the skip-gram model as

$$P(w_o \mid w_c) = \prod_{j=1}^{L(w_o)-1} \sigma\left([\![n(w_o, j+1) = \text{leftChild}(n(w_o, j))]\!] \cdot \mathbf{u}_{n(w_o,j)}^\top \mathbf{v}_c\right), \tag{14.2.9}$$

Here the σ function has the same definition as the sigmoid activation function, and leftChild(n) is the left child node of node n. If x is true, $[\![x]\!] = 1$; otherwise $[\![x]\!] = -1$. Now, we will compute the conditional probability of generating word w_3 based on the given word w_c in Figure 10.3. We need to find the inner product of word vector $\mathbf{v}_c$ (for word w_c) and each non-leaf node vector on the path from the root node to w_3. Because, in the binary tree, the path from the root node to leaf node w_3 needs to be traversed left, right, and left again (the path with the bold line in Figure 10.3), we get

$$P(w_3 \mid w_c) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_c). \tag{14.2.10}$$

Because $\sigma(x) + \sigma(-x) = 1$, the condition that the sum of the conditional probability of any word generated based on the given central target word w_c in dictionary $\mathcal{V}$ be 1 will also suffice:

$$\sum_{w \in \mathcal{V}} P(w \mid w_c) = 1. \tag{14.2.11}$$

In addition, because the order of magnitude for $L(w_o) - 1$ is $\mathcal{O}(\log_2 |\mathcal{V}|)$, when the size of dictionary $\mathcal{V}$ is large, the computational overhead for each step in the hierarchical softmax training is greatly reduced compared to situations where we do not use approximate training.

Summary

- Negative sampling constructs the loss function by considering independent events that contain both positive and negative examples. The gradient computational overhead for each step in the training process is linearly related to the number of noise words we sample.
- Hierarchical softmax uses a binary tree and constructs the loss function based on the path from the root node to the leaf node. The gradient computational overhead for each step in the training process is related to the logarithm of the dictionary size.

Exercises

1. Before reading the next section, think about how we should sample noise words in negative sampling.
2. What makes the last formula in this section hold?
3. How can we apply negative sampling and hierarchical softmax in the skip-gram model?

14.3 The Dataset for Word2vec

In this section, we will introduce how to preprocess a dataset with negative sampling Section 14.2 and load into minibatches for word2vec training. The dataset we use is Penn Tree Bank (PTB)[219], which is a small but commonly-used corpus. It takes samples from Wall Street Journal articles and includes training sets, validation sets, and test sets.

First, import the packages and modules required for the experiment.

```python
import d2l
import math
from mxnet import gluon, np
import random
```

14.3.1 Reading and Preprocessing the Dataset

This dataset has already been preprocessed. Each line of the dataset acts as a sentence. All the words in a sentence are separated by spaces. In the word embedding task, each word is a token.

```python
# Saved in the d2l package for later use
d2l.DATA_HUB['ptb'] = (d2l.DATA_URL+'ptb.zip',
                       '319d85e578af0cdc590547f26231e4e31cdf1e42')
def read_ptb():
    data_dir = d2l.download_extract('ptb')
```

(continues on next page)

[219] https://catalog.ldc.upenn.edu/LDC99T42

(continued from previous page)

```
    with open(data_dir+'ptb.train.txt') as f:
        raw_text = f.read()
    return [line.split() for line in raw_text.split('\n')]

sentences = read_ptb()
'# sentences: %d' % len(sentences)
```

```
'# sentences: 42069'
```

Next we build a vocabulary with words appeared not greater than 10 times mapped into a "<unk>" token. Note that the preprocessed PTB data also contains "<unk>" tokens presenting rare words.

```
vocab = d2l.Vocab(sentences, min_freq=10)
'vocab size: %d' % len(vocab)
```

```
'vocab size: 6719'
```

14.3.2 Subsampling

In text data, there are generally some words that appear at high frequencies, such "the", "a", and "in" in English. Generally speaking, in a context window, it is better to train the word embedding model when a word (such as "chip") and a lower-frequency word (such as "microprocessor") appear at the same time, rather than when a word appears with a higher-frequency word (such as "the"). Therefore, when training the word embedding model, we can perform subsampling[2] on the words. Specifically, each indexed word w_i in the dataset will drop out at a certain probability. The dropout probability is given as:

$$P(w_i) = \max\left(1 - \sqrt{\frac{t}{f(w_i)}}, 0\right), \tag{14.3.1}$$

Here, $f(w_i)$ is the ratio of the instances of word w_i to the total number of words in the dataset, and the constant t is a hyperparameter (set to 10^{-4} in this experiment). As we can see, it is only possible to drop out the word w_i in subsampling when $f(w_i) > t$. The higher the word's frequency, the higher its dropout probability.

```
# Saved in the d2l package for later use
def subsampling(sentences, vocab):
    # Map low frequency words into <unk>
    sentences = [[vocab.idx_to_token[vocab[tk]] for tk in line]
                 for line in sentences]
    # Count the frequency for each word
    counter = d2l.count_corpus(sentences)
    num_tokens = sum(counter.values())

    # Return True if to keep this token during subsampling
    def keep(token):
        return(random.uniform(0, 1) <
               math.sqrt(1e-4 / counter[token] * num_tokens))
```

(continues on next page)

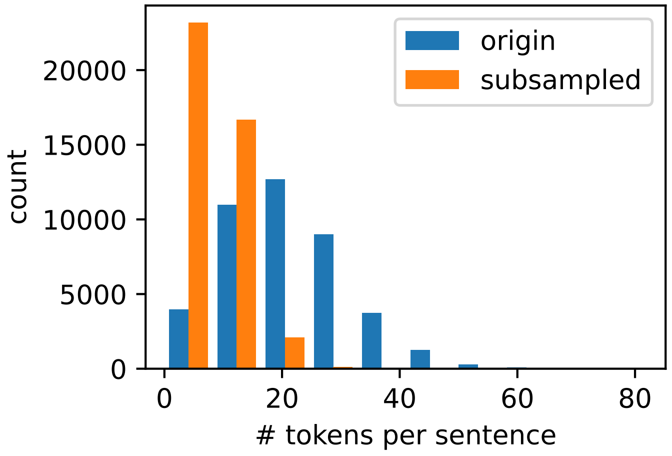
(continued from previous page)

```
    # Now do the subsampling
    return [[tk for tk in line if keep(tk)] for line in sentences]

subsampled = subsampling(sentences, vocab)
```

Compare the sequence lengths before and after sampling, we can see subsampling significantly reduced the sequence length.

```
d2l.set_figsize((3.5, 2.5))
d2l.plt.hist([[len(line) for line in sentences],
              [len(line) for line in subsampled]])
d2l.plt.xlabel('# tokens per sentence')
d2l.plt.ylabel('count')
d2l.plt.legend(['origin', 'subsampled']);
```



For individual tokens, the sampling rate of the high-frequency word “the” is less than 1/20.

```
def compare_counts(token):
    return '# of "%s": before=%d, after=%d' % (token, sum(
        [line.count(token) for line in sentences]), sum(
        [line.count(token) for line in subsampled]))

compare_counts('the')
```

```
'# of "the": before=50770, after=2147'
```

But the low-frequency word “join” is completely preserved.

```
compare_counts('join')
```

```
'# of "join": before=45, after=45'
```

Last, we map each token into an index to construct the corpus.

```
corpus = [vocab[line] for line in subsampled]
corpus[0:3]
```

```
[[], [392, 2132, 275], [5464, 3080, 1595]]
```

14.3.3 Loading the Dataset

Next we read the corpus with token indicies into data batches for training.

Extracting Central Target Words and Context Words

We use words with a distance from the central target word not exceeding the context window size as the context words of the given center target word. The following definition function extracts all the central target words and their context words. It uniformly and randomly samples an integer to be used as the context window size between integer 1 and the `max_window_size` (maximum context window).

```
# Saved in the d2l package for later use
def get_centers_and_contexts(corpus, max_window_size):
    centers, contexts = [], []
    for line in corpus:
        # Each sentence needs at least 2 words to form a
        # "central target word - context word" pair
        if len(line) < 2:
            continue
        centers += line
        for i in range(len(line)):  # Context window centered at i
            window_size = random.randint(1, max_window_size)
            indices = list(range(max(0, i - window_size),
                                 min(len(line), i + 1 + window_size)))
            # Exclude the central target word from the context words
            indices.remove(i)
            contexts.append([line[idx] for idx in indices])
    return centers, contexts
```

Next, we create an artificial dataset containing two sentences of 7 and 3 words, respectively. Assume the maximum context window is 2 and print all the central target words and their context words.

```
tiny_dataset = [list(range(7)), list(range(7, 10))]
print('dataset', tiny_dataset)
for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):
    print('center', center, 'has contexts', context)
```

```
dataset [[0, 1, 2, 3, 4, 5, 6], [7, 8, 9]]
center 0 has contexts [1, 2]
center 1 has contexts [0, 2, 3]
center 2 has contexts [1, 3]
center 3 has contexts [1, 2, 4, 5]
```

(continues on next page)

(continued from previous page)

```
center 4 has contexts [3, 5]
center 5 has contexts [3, 4, 6]
center 6 has contexts [5]
center 7 has contexts [8]
center 8 has contexts [7, 9]
center 9 has contexts [8]
```

We set the maximum context window size to 5. The following extracts all the central target words and their context words in the dataset.

```
all_centers, all_contexts = get_centers_and_contexts(corpus, 5)
'# center-context pairs: %d' % len(all_centers)
```

```
'# center-context pairs: 353210'
```

Negative Sampling

We use negative sampling for approximate training. For a central and context word pair, we randomly sample K noise words ($K = 5$ in the experiment). According to the suggestion in the Word2vec paper, the noise word sampling probability $P(w)$ is the ratio of the word frequency of w to the total word frequency raised to the power of 0.75 [2].

We first define a class to draw a candidate according to the sampling weights. It caches a 10000 size random number bank instead of calling `random.choices` every time.

```
# Saved in the d2l package for later use
class RandomGenerator(object):
    """Draw a random int in [0, n] according to n sampling weights"""
    def __init__(self, sampling_weights):
        self.population = list(range(len(sampling_weights)))
        self.sampling_weights = sampling_weights
        self.candidates = []
        self.i = 0

    def draw(self):
        if self.i == len(self.candidates):
            self.candidates = random.choices(
                self.population, self.sampling_weights, k=10000)
            self.i = 0
        self.i += 1
        return self.candidates[self.i-1]

generator = RandomGenerator([2, 3, 4])
[generator.draw() for _ in range(10)]
```

```
[1, 2, 2, 0, 0, 2, 2, 0, 1, 2]
```

```
# Saved in the d2l package for later use
def get_negatives(all_contexts, corpus, K):
```

(continues on next page)

(continued from previous page)

```
    counter = d2l.count_corpus(corpus)
    sampling_weights = [counter[i]**0.75 for i in range(len(counter))]
    all_negatives, generator = [], RandomGenerator(sampling_weights)
    for contexts in all_contexts:
        negatives = []
        while len(negatives) < len(contexts) * K:
            neg = generator.draw()
            # Noise words cannot be context words
            if neg not in contexts:
                negatives.append(neg)
        all_negatives.append(negatives)
    return all_negatives

all_negatives = get_negatives(all_contexts, corpus, 5)
```

Reading into Batches

We extract all central target words `all_centers`, and the context words `all_contexts` and noise words `all_negatives` of each central target word from the dataset. We will read them in random minibatches.

In a minibatch of data, the i^{th} example includes a central word and its corresponding n_i context words and m_i noise words. Since the context window size of each example may be different, the sum of context words and noise words, $n_i + m_i$, will be different. When constructing a minibatch, we concatenate the context words and noise words of each example, and add 0s for padding until the length of the concatenations are the same, that is, the length of all concatenations is $\max_i n_i + m_i$(`max_len`). In order to avoid the effect of padding on the loss function calculation, we construct the mask variable `masks`, each element of which corresponds to an element in the concatenation of context and noise words, `contexts_negatives`. When an element in the variable `contexts_negatives` is a padding, the element in the mask variable `masks` at the same position will be 0. Otherwise, it takes the value 1. In order to distinguish between positive and negative examples, we also need to distinguish the context words from the noise words in the `contexts_negatives` variable. Based on the construction of the mask variable, we only need to create a label variable `labels` with the same shape as the `contexts_negatives` variable and set the elements corresponding to context words (positive examples) to 1, and the rest to 0.

Next, we will implement the minibatch reading function `batchify`. Its minibatch input `data` is a list whose length is the batch size, each element of which contains central target words `center`, context words `context`, and noise words `negative`. The minibatch data returned by this function conforms to the format we need, for example, it includes the mask variable.

```
# Saved in the d2l package for later use
def batchify(data):
    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, masks, labels = [], [], [], []
    for center, context, negative in data:
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [context + negative + [0] * (max_len - cur_len)]
        masks += [[1] * cur_len + [0] * (max_len - cur_len)]
        labels += [[1] * len(context) + [0] * (max_len - len(context))]
```

(continues on next page)

(continued from previous page)

```
    return (np.array(centers).reshape(-1, 1), np.array(contexts_negatives),
            np.array(masks), np.array(labels))
```

Construct two simple examples:

```
x_1 = (1, [2, 2], [3, 3, 3, 3])
x_2 = (1, [2, 2, 2], [3, 3])
batch = batchify((x_1, x_2))

names = ['centers', 'contexts_negatives', 'masks', 'labels']
for name, data in zip(names, batch):
    print(name, '=', data)
```

```
centers = [[1.]
 [1.]]
contexts_negatives = [[2. 2. 3. 3. 3. 3.]
 [2. 2. 2. 3. 3. 0.]]
masks = [[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 0.]]
labels = [[1. 1. 0. 0. 0. 0.]
 [1. 1. 1. 0. 0. 0.]]
```

We use the `batchify` function just defined to specify the minibatch reading method in the `DataLoader` instance.

14.3.4 Putting All Things Together

Last, we define the `load_data_ptb` function that read the PTB dataset and return the data loader.

```
# Saved in the d2l package for later use
def load_data_ptb(batch_size, max_window_size, num_noise_words):
    sentences = read_ptb()
    vocab = d2l.Vocab(sentences, min_freq=10)
    subsampled = subsampling(sentences, vocab)
    corpus = [vocab[line] for line in subsampled]
    all_centers, all_contexts = get_centers_and_contexts(
        corpus, max_window_size)
    all_negatives = get_negatives(all_contexts, corpus, num_noise_words)
    dataset = gluon.data.ArrayDataset(
        all_centers, all_contexts, all_negatives)
    data_iter = gluon.data.DataLoader(dataset, batch_size, shuffle=True,
                                      batchify_fn=batchify)
    return data_iter, vocab
```

Let's print the first minibatch of the data iterator.

```
data_iter, vocab = load_data_ptb(512, 5, 5)
for batch in data_iter:
    for name, data in zip(names, batch):
        print(name, 'shape:', data.shape)
    break
```

```
centers shape: (512, 1)
contexts_negatives shape: (512, 60)
masks shape: (512, 60)
labels shape: (512, 60)
```

Summary

- Subsampling attempts to minimize the impact of high-frequency words on the training of a word embedding model.
- We can pad examples of different lengths to create minibatches with examples of all the same length and use mask variables to distinguish between padding and non-padding elements, so that only non-padding elements participate in the calculation of the loss function.

Exercises

1. We use the `batchify` function to specify the minibatch reading method in the `DataLoader` instance and print the shape of each variable in the first batch read. How should these shapes be calculated?

14.4 Implementation of Word2vec

In this section, we will train a skip-gram model defined in Section 14.1.

First, import the packages and modules required for the experiment, and load the PTB dataset.

```
import d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
npx.set_np()

batch_size, max_window_size, num_noise_words = 512, 5, 5
data_iter, vocab = d2l.load_data_ptb(512, 5, 5)
```

14.4.1 The Skip-Gram Model

We will implement the skip-gram model by using embedding layers and minibatch multiplication. These methods are also often used to implement other natural language processing applications.

Embedding Layer

The layer in which the obtained word is embedded is called the embedding layer, which can be obtained by creating an `nn.Embedding` instance in Gluon. The weight of the embedding layer is a matrix whose number of rows is the dictionary size (`input_dim`) and whose number of columns is the dimension of each word vector (`output_dim`). We set the dictionary size to 20 and the word vector dimension to 4.

```
embed = nn.Embedding(input_dim=20, output_dim=4)
embed.initialize()
embed.weight
```

```
Parameter embedding0_weight (shape=(20, 4), dtype=float32)
```

The input of the embedding layer is the index of the word. When we enter the index i of a word, the embedding layer returns the i^{th} row of the weight matrix as its word vector. Below we enter an index of shape $(2, 3)$ into the embedding layer. Because the dimension of the word vector is 4, we obtain a word vector of shape $(2, 3, 4)$.

```
x = np.array([[1, 2, 3], [4, 5, 6]])
embed(x)
```

```
array([[[ 0.01438687,  0.05011239,  0.00628365,  0.04861524],
        [-0.01068833,  0.01729892,  0.02042518, -0.01618656],
        [-0.00873779, -0.02834515,  0.05484822, -0.06206018]],

       [[ 0.06491279, -0.03182812, -0.01631819, -0.00312688],
        [ 0.0408415 ,  0.04370362,  0.00404529, -0.0028032 ],
        [ 0.00952624, -0.01501013,  0.05958354,  0.04705103]]])
```

Minibatch Multiplication

We can multiply the matrices in two minibatches one by one, by the minibatch multiplication operation `batch_dot`. Suppose the first batch contains n matrices $\mathbf{X}_1, \ldots, \mathbf{X}_n$ with a shape of $a \times b$, and the second batch contains n matrices $\mathbf{Y}_1, \ldots, \mathbf{Y}_n$ with a shape of $b \times c$. The output of matrix multiplication on these two batches are n matrices $\mathbf{X}_1\mathbf{Y}_1, \ldots, \mathbf{X}_n\mathbf{Y}_n$ with a shape of $a \times c$. Therefore, given two `ndarray`s of shape (n, a, b) and (n, b, c), the shape of the minibatch multiplication output is (n, a, c).

```
X = np.ones((2, 1, 4))
Y = np.ones((2, 4, 6))
npx.batch_dot(X, Y).shape
```

```
(2, 1, 6)
```

Skip-gram Model Forward Calculation

In forward calculation, the input of the skip-gram model contains the central target word index `center` and the concatenated context and noise word index `contexts_and_negatives`. In which, the `center` variable has the shape (batch size, 1), while the `contexts_and_negatives` variable has the shape (batch size, `max_len`). These two variables are first transformed from word indexes to word vectors by the word embedding layer, and then the output of shape (batch size, 1, `max_len`) is obtained by minibatch multiplication. Each element in the output is the inner product of the central target word vector and the context word vector or noise word vector.

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = npx.batch_dot(v, u.swapaxes(1, 2))
    return pred
```

Verify that the output shape should be (batch size, 1, `max_len`).

```
skip_gram(np.ones((2, 1)), np.ones((2, 4)), embed, embed).shape
```

```
(2, 1, 4)
```

14.4.2 Training

Before training the word embedding model, we need to define the loss function of the model.

Binary Cross Entropy Loss Function

According to the definition of the loss function in negative sampling, we can directly use Gluon's binary cross-entropy loss function `SigmoidBinaryCrossEntropyLoss`.

```
loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()
```

It is worth mentioning that we can use the mask variable to specify the partial predicted value and label that participate in loss function calculation in the minibatch: when the mask is 1, the predicted value and label of the corresponding position will participate in the calculation of the loss function; When the mask is 0, the predicted value and label of the corresponding position do not participate in the calculation of the loss function. As we mentioned earlier, mask variables can be used to avoid the effect of padding on loss function calculations.

Given two identical examples, different masks lead to different loss values.

```
pred = np.array([[.5]*4]*2)
label = np.array([[1, 0, 1, 0]]*2)
mask = np.array([[1, 1, 1, 1], [1, 1, 0, 0]])
loss(pred, label, mask)
```

```
array([0.724077 , 0.3620385])
```

We can normalize the loss in each example due to various lengths in each example.

```
loss(pred, label, mask) / mask.sum(axis=1) * mask.shape[1]
```

```
array([0.724077, 0.724077])
```

Initializing Model Parameters

We construct the embedding layers of the central and context words, respectively, and set the hyperparameter word vector dimension `embed_size` to 100.

```
embed_size = 100
net = nn.Sequential()
net.add(nn.Embedding(input_dim=len(vocab), output_dim=embed_size),
        nn.Embedding(input_dim=len(vocab), output_dim=embed_size))
```

Training

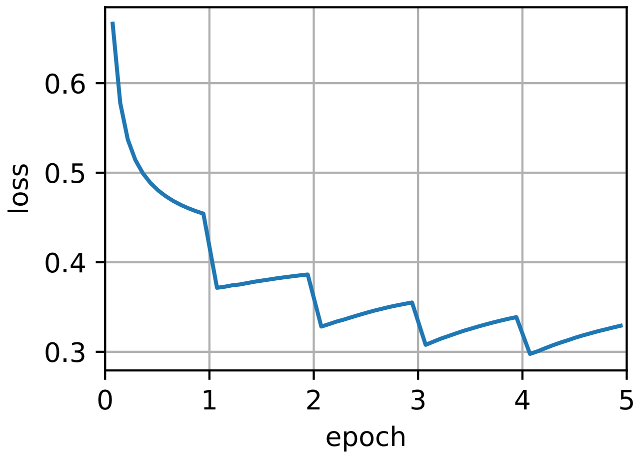
The training function is defined below. Because of the existence of padding, the calculation of the loss function is slightly different compared to the previous training functions.

```
def train(net, data_iter, lr, num_epochs, ctx=d2l.try_gpu()):
    net.initialize(ctx=ctx, force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs])
    for epoch in range(num_epochs):
        timer = d2l.Timer()
        metric = d2l.Accumulator(2)  # loss_sum, num_tokens
        for i, batch in enumerate(data_iter):
            center, context_negative, mask, label = [
                data.as_in_context(ctx) for data in batch]
            with autograd.record():
                pred = skip_gram(center, context_negative, net[0], net[1])
                l = (loss(pred.reshape(label.shape), label, mask)
                     / mask.sum(axis=1) * mask.shape[1])
            l.backward()
            trainer.step(batch_size)
            metric.add(l.sum(), l.size)
            if (i+1) % 50 == 0:
                animator.add(epoch+(i+1)/len(data_iter),
                             (metric[0]/metric[1],))
    print('loss %.3f, %d tokens/sec on %s ' % (
        metric[0]/metric[1], metric[1]/timer.stop(), ctx))
```

Now, we can train a skip-gram model using negative sampling.

```
lr, num_epochs = 0.01, 5
train(net, data_iter, lr, num_epochs)
```

```
loss 0.330, 27378 tokens/sec on gpu(0)
```



14.4.3 Applying the Word Embedding Model

After training the word embedding model, we can represent similarity in meaning between words based on the cosine similarity of two word vectors. As we can see, when using the trained word embedding model, the words closest in meaning to the word "chip" are mostly related to chips.

```
def get_similar_tokens(query_token, k, embed):
    W = embed.weight.data()
    x = W[vocab[query_token]]
    # Compute the cosine similarity. Add 1e-9 for numerical stability
    cos = np.dot(W, x) / np.sqrt(np.sum(W * W, axis=1) * np.sum(x * x) + 1e-9)
    topk = npx.topk(cos, k=k+1, ret_typ='indices').asnumpy().astype('int32')
    for i in topk[1:]:  # Remove the input words
        print('cosine sim=%.3f: %s' % (cos[i], (vocab.idx_to_token[i])))

get_similar_tokens('chip', 3, net[0])
```

```
cosine sim=0.584: intel
cosine sim=0.558: computer
cosine sim=0.524: mips
```

Summary

- We can use Gluon to train a skip-gram model through negative sampling.

Exercises

1. Set `sparse_grad=True` when creating an instance of `nn.Embedding`. Does it accelerate training? Look up MXNet documentation to learn the meaning of this argument.
2. Try to find synonyms for other words.
3. Tune the hyper-parameters and observe and analyze the experimental results.
4. When the dataset is large, we usually sample the context words and the noise words for the central target word in the current minibatch only when updating the model parameters. In other words, the same central target word may have different context words or noise words in different epochs. What are the benefits of this sort of training? Try to implement this training method.

14.5 Subword Embedding (fastText)

English words usually have internal structures and formation methods. For example, we can deduce the relationship between "dog", "dogs", and "dogcatcher" by their spelling. All these words have the same root, "dog", but they use different suffixes to change the meaning of the word. Moreover, this association can be extended to other words. For example, the relationship between "dog" and "dogs" is just like the relationship between "cat" and "cats". The relationship between "boy" and "boyfriend" is just like the relationship between "girl" and "girlfriend". This characteristic is not unique to English. In French and Spanish, a lot of verbs can have more than 40 different forms depending on the context. In Finnish, a noun may have more than 15 forms. In fact, morphology, which is an important branch of linguistics, studies the internal structure and formation of words.

In word2vec, we did not directly use morphology information. In both the skip-gram model and continuous bag-of-words model, we use different vectors to represent words with different forms. For example, "dog" and "dogs" are represented by two different vectors, while the relationship between these two vectors is not directly represented in the model. In view of this, fastText (Bojanowski et al., 2017) proposes the method of subword embedding, thereby attempting to introduce morphological information in the skip-gram model in word2vec.

In fastText, each central word is represented as a collection of subwords. Below we use the word "where" as an example to understand how subwords are formed. First, we add the special characters "<" and ">" at the beginning and end of the word to distinguish the subwords used as prefixes and suffixes. Then, we treat the word as a sequence of characters to extract the n-grams. For example, when $n = 3$, we can get all subwords with a length of 3:

$$\text{"<wh", "whe", "her", "ere", "re>",} \tag{14.5.1}$$

and the special subword "<where>".

In fastText, for a word w, we record the union of all its subwords with length of 3 to 6 and special subwords as $\mathcal{G}_w$. Thus, the dictionary is the union of the collection of subwords of all words. Assume the vector of the subword g in the dictionary is $\mathbf{z}_g$. Then, the central word vector $\mathbf{u}_w$ for the word w in the skip-gram model can be expressed as

$$\mathbf{u}_w = \sum_{g\in\mathcal{G}_w} \mathbf{z}_g. \tag{14.5.2}$$

The rest of the fastText process is consistent with the skip-gram model, so it is not repeated here. As we can see, compared with the skip-gram model, the dictionary in fastText is larger, resulting in more model parameters. Also, the vector of one word requires the summation of all subword vectors, which results in higher computation complexity. However, we can obtain better vectors for more uncommon complex words, even words not existing in the dictionary, by looking at other words with similar structures.

Summary

- FastText proposes a subword embedding method. Based on the skip-gram model in word2vec, it represents the central word vector as the sum of the subword vectors of the word.
- Subword embedding utilizes the principles of morphology, which usually improves the quality of representations of uncommon words.

Exercises

1. When there are too many subwords (for example, 6 words in English result in about 3×10^8 combinations), what problems arise? Can you think of any methods to solve them? Hint: Refer to the end of section 3.2 of the fastText paper[1].
2. How can you design a subword embedding model based on the continuous bag-of-words model?

14.6 Word Embedding with Global Vectors (GloVe)

First, we should review the skip-gram model in word2vec. The conditional probability $P(w_j \mid w_i)$ expressed in the skip-gram model using the softmax operation will be recorded as q_{ij}, that is:

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k\in\mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)}, \tag{14.6.1}$$

where $\mathbf{v}_i$ and $\mathbf{u}_i$ are the vector representations of word w_i of index i as the center word and context word respectively, and $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$ is the vocabulary index set.

For word w_i, it may appear in the dataset for multiple times. We collect all the context words every time when w_i is a center word and keep duplicates, denoted as multiset $\mathcal{C}_i$. The number of an element in a multiset is called the multiplicity of the element. For instance, suppose that word w_i appears twice in the dataset: the context windows when these two w_i become center words in the text sequence contain context word indices $2, 1, 5, 2$ and $2, 3, 2, 1$. Then, multiset $\mathcal{C}_i = \{1, 1, 2, 2, 2, 2, 3, 5\}$, where multiplicity of element 1 is 2, multiplicity of element 2 is 4, and multiplicities of elements 3 and 5 are both 1. Denote multiplicity of element j in multiset $\mathcal{C}_i$ as x_{ij}: it is the number of word w_j in all the context windows for center word w_i in the entire dataset. As a result, the loss function of the skip-gram model can be expressed in a different way:

$$-\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{V}} x_{ij} \log\, q_{ij}. \tag{14.6.2}$$

We add up the number of all the context words for the central target word w_i to get x_i, and record the conditional probability x_{ij}/x_i for generating context word w_j based on central target word w_i as p_{ij}. We can rewrite the loss function of the skip-gram model as

$$-\sum_{i\in\mathcal{V}} x_i \sum_{j\in\mathcal{V}} p_{ij} \log\, q_{ij}. \tag{14.6.3}$$

In the formula above, $\sum_{j\in\mathcal{V}} p_{ij} \log\, q_{ij}$ computes the conditional probability distribution p_{ij} for context word generation based on the central target word w_i and the cross-entropy of conditional probability distribution q_{ij} predicted by the model. The loss function is weighted using the sum of the number of context words with the central target word w_i. If we minimize the loss function from the formula above, we will be able to allow the predicted conditional probability distribution to approach as close as possible to the true conditional probability distribution.

However, although the most common type of loss function, the cross-entropy loss function is sometimes not a good choice. On the one hand, as we mentioned in Section 14.2 the cost of letting the model prediction q_{ij} become the legal probability distribution has the sum of all items in the entire dictionary in its denominator. This can easily lead to excessive computational overhead. On the other hand, there are often a lot of uncommon words in the dictionary, and they appear rarely in the dataset. In the cross-entropy loss function, the final prediction of the conditional probability distribution on a large number of uncommon words is likely to be inaccurate.

14.6.1 The GloVe Model

To address this, GloVe (Pennington et al., 2014), a word embedding model that came after word2vec, adopts square loss and makes three changes to the skip-gram model based on this loss.

1. Here, we use the non-probability distribution variables $p'_{ij} = x_{ij}$ and $q'_{ij} = \exp(\mathbf{u}_j^\top \mathbf{v}_i)$ and take their logs. Therefore, we get the square loss $\left(\log\, p'_{ij} - \log\, q'_{ij}\right)^2 = \left(\mathbf{u}_j^\top \mathbf{v}_i - \log\, x_{ij}\right)^2$.
2. We add two scalar model parameters for each word w_i: the bias terms b_i (for central target words) and c_i(for context words).
3. Replace the weight of each loss with the function $h(x_{ij})$. The weight function $h(x)$ is a monotone increasing function with the range $[0, 1].

Therefore, the goal of GloVe is to minimize the loss function.

$$\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{V}} h(x_{ij}) \left(\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log\, x_{ij}\right)^2. \tag{14.6.4}$$

Here, we have a suggestion for the choice of weight function $h(x)$: when $x < c$ (e.g $c = 100$), make $h(x) = (x/c)^\alpha$ (e.g $\alpha = 0.75$), otherwise make $h(x) = 1$. Because $h(0) = 0$, the squared loss term for $x_{ij} = 0$ can be simply ignored. When we use minibatch SGD for training, we conduct random sampling to get a non-zero minibatch x_{ij} from each timestep and compute the gradient to update the model parameters. These non-zero x_{ij} are computed in advance based on the entire dataset and they contain global statistics for the dataset. Therefore, the name GloVe is taken from "Global Vectors".

Notice that if word w_i appears in the context window of word w_j, then word w_j will also appear in the context window of word w_i. Therefore, $x_{ij} = x_{ji}$. Unlike word2vec, GloVe fits the symmetric $\log x_{ij}$ in lieu of the asymmetric conditional probability p_{ij}. Therefore, the central target word vector and context word vector of any word are equivalent in GloVe. However, the two sets of word vectors that are learned by the same word may be different in the end due to different initialization values. After learning all the word vectors, GloVe will use the sum of the central target word vector and the context word vector as the final word vector for the word.

14.6.2 Understanding GloVe from Conditional Probability Ratios

We can also try to understand GloVe word embedding from another perspective. We will continue the use of symbols from earlier in this section, $P(w_j \mid w_i)$ represents the conditional probability of generating context word w_j with central target word w_i in the dataset, and it will be recorded as p_{ij}. From a real example from a large corpus, here we have the following two sets of conditional probabilities with "ice" and "steam" as the central target words and the ratio between them:

w_k=	"solid"	"gas"	"water"	"fashion"
$p_1 = P(w_k \mid$ "ice" $)$	0.00019	0.000066	0.003	0.000017
$p_2 = P(w_k \mid$ "steam" $)$	0.000022	0.00078	0.0022	0.000018
p_1/p_2	8.9	0.085	1.36	0.96

We will be able to observe phenomena such as:

- For a word w_k that is related to "ice" but not to "steam", such as $w_k =$"solid", we would expect a larger conditional probability ratio, like the value 8.9 in the last row of the table above.
- For a word w_k that is related to "steam" but not to "ice", such as $w_k =$"gas", we would expect a smaller conditional probability ratio, like the value 0.085 in the last row of the table above.
- For a word w_k that is related to both "ice" and "steam", such as $w_k =$"water", we would expect a conditional probability ratio close to 1, like the value 1.36 in the last row of the table above.
- For a word w_k that is related to neither "ice" or "steam", such as $w_k =$"fashion", we would expect a conditional probability ratio close to 1, like the value 0.96 in the last row of the table above.

We can see that the conditional probability ratio can represent the relationship between different words more intuitively. We can construct a word vector function to fit the conditional probability ratio more effectively. As we know, to obtain any ratio of this type requires three words w_i, w_j, and w_k. The conditional probability ratio with w_i as the central target word is p_{ij}/p_{ik}. We can find a function that uses word vectors to fit this conditional probability ratio.

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) \approx \frac{p_{ij}}{p_{ik}}. \tag{14.6.5}$$

The possible design of function f here will not be unique. We only need to consider a more reasonable possibility. Notice that the conditional probability ratio is a scalar, we can limit f to be a scalar function: $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = f\left((\mathbf{u}_j - \mathbf{u}_k)^\top \mathbf{v}_i\right)$. After exchanging index j with k, we will be able to see that function f satisfies the condition $f(x)f(-x) = 1$, so one possibility could be $f(x) = \exp(x)$. Thus:

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = \frac{\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right)}{\exp\left(\mathbf{u}_k^\top \mathbf{v}_i\right)} \approx \frac{p_{ij}}{p_{ik}}. \tag{14.6.6}$$

One possibility that satisfies the right side of the approximation sign is $\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right) \approx \alpha p_{ij}$, where α is a constant. Considering that $p_{ij} = x_{ij}/x_i$, after taking the logarithm we get $\mathbf{u}_j^\top \mathbf{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$. We use additional bias terms to fit $-\log \alpha + \log x_i$, such as the central target word bias term b_i and context word bias term c_j:

$$\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j \approx \log(x_{ij}). \tag{14.6.7}$$

By taking the square error and weighting the left and right sides of the formula above, we can get the loss function of GloVe.

Summary

- In some cases, the cross-entropy loss function may have a disadvantage. GloVe uses squared loss and the word vector to fit global statistics computed in advance based on the entire dataset.
- The central target word vector and context word vector of any word are equivalent in GloVe.

Exercises

1. If a word appears in the context window of another word, how can we use the distance between them in the text sequence to redesign the method for computing the conditional probability p_{ij}? Hint: See section 4.2 from the paper GloVe (Pennington et al., 2014).
2. For any word, will its central target word bias term and context word bias term be equivalent to each other in GloVe? Why?

14.7 Finding Synonyms and Analogies

In Section 14.4 we trained a word2vec word embedding model on a small-scale dataset and searched for synonyms using the cosine similarity of word vectors. In practice, word vectors pre-trained on a large-scale corpus can often be applied to downstream natural language processing tasks. This section will demonstrate how to use these pre-trained word vectors to find synonyms and analogies. We will continue to apply pre-trained word vectors in subsequent sections.

14.7.1 Using Pre-Trained Word Vectors

MXNet's `contrib.text` package provides functions and classes related to natural language processing (see the GluonNLP[224] tool package for more details). Next, let's check out names of the provided pre-trained word embeddings.

```
from mxnet import np, npx
from mxnet.contrib import text
npx.set_np()

text.embedding.get_pretrained_file_names().keys()
```

```
dict_keys(['glove', 'fasttext'])
```

Given the name of the word embedding, we can see which pre-trained models are provided by the word embedding. The word vector dimensions of each model may be different or obtained by pre-training on different datasets.

```
print(text.embedding.get_pretrained_file_names('glove'))
```

```
['glove.42B.300d.txt', 'glove.6B.50d.txt', 'glove.6B.100d.txt', 'glove.6B.200d.txt', 'glove.
↪6B.300d.txt', 'glove.840B.300d.txt', 'glove.twitter.27B.25d.txt', 'glove.twitter.27B.50d.
↪txt', 'glove.twitter.27B.100d.txt', 'glove.twitter.27B.200d.txt']
```

The general naming conventions for pre-trained GloVe models are "model.(dataset.)number of words in dataset.word vector dimension.txt". For more information, please refer to the GloVe and fastText project sites [2, 3]. Below, we use a 50-dimensional GloVe word vector based on Wikipedia subset pre-training. The corresponding word vector is automatically downloaded the first time we create a pre-trained word vector instance.

```
glove_6b50d = text.embedding.create(
    'glove', pretrained_file_name='glove.6B.50d.txt')
```

Print the dictionary size. The dictionary contains $400,000$ words and a special unknown token.

```
len(glove_6b50d)
```

```
400001
```

We can use a word to get its index in the dictionary, or we can get the word from its index.

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

```
(3367, 'beautiful')
```

[224] https://gluon-nlp.mxnet.io/

14.7.2 Applying Pre-Trained Word Vectors

Below, we demonstrate the application of pre-trained word vectors, using GloVe as an example.

Finding Synonyms

Here, we re-implement the algorithm used to search for synonyms by cosine similarity introduced in Section 14.1

In order to reuse the logic for seeking the k nearest neighbors when seeking analogies, we encapsulate this part of the logic separately in the `knn` (k-nearest neighbors) function.

```
def knn(W, x, k):
    # The added 1e-9 is for numerical stability
    cos = np.dot(W, x.reshape(-1,)) / (
        np.sqrt(np.sum(W * W, axis=1) + 1e-9) * np.sqrt((x * x).sum()))
    topk = npx.topk(cos, k=k, ret_typ='indices')
    return topk, [cos[int(i)] for i in topk]
```

Then, we search for synonyms by pre-training the word vector instance `embed`.

```
def get_similar_tokens(query_token, k, embed):
    topk, cos = knn(embed.idx_to_vec,
                    embed.get_vecs_by_tokens([query_token]), k+1)
    for i, c in zip(topk[1:], cos[1:]):  # Remove input words
        print('cosine sim=%.3f: %s' % (c, (embed.idx_to_token[int(i)])))
```

The dictionary of pre-trained word vector instance `glove_6b50d` already created contains 400,000 words and a special unknown token. Excluding input words and unknown words, we search for the three words that are the most similar in meaning to "chip".

```
get_similar_tokens('chip', 3, glove_6b50d)
```

```
cosine sim=0.856: chips
cosine sim=0.749: intel
cosine sim=0.749: electronics
```

Next, we search for the synonyms of "baby" and "beautiful".

```
get_similar_tokens('baby', 3, glove_6b50d)
```

```
cosine sim=0.839: babies
cosine sim=0.800: boy
cosine sim=0.792: girl
```

```
get_similar_tokens('beautiful', 3, glove_6b50d)
```

```
cosine sim=0.921: lovely
cosine sim=0.893: gorgeous
cosine sim=0.830: wonderful
```

Finding Analogies

In addition to seeking synonyms, we can also use the pre-trained word vector to seek the analogies between words. For example, "man":"woman"::"son":"daughter" is an example of analogy, "man" is to "woman" as "son" is to "daughter". The problem of seeking analogies can be defined as follows: for four words in the analogical relationship $a : b :: c : d$, given the first three words, a, b and c, we want to find d. Assume the word vector for the word w is vec(w). To solve the analogy problem, we need to find the word vector that is most similar to the result vector of vec(c) + vec(b) − vec(a).

```python
def get_analogy(token_a, token_b, token_c, embed):
    vecs = embed.get_vecs_by_tokens([token_a, token_b, token_c])
    x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = knn(embed.idx_to_vec, x, 1)
    return embed.idx_to_token[int(topk[0])]  # Remove unknown words
```

Verify the "male-female" analogy.

```python
get_analogy('man', 'woman', 'son', glove_6b50d)
```

```
'daughter'
```

"Capital-country" analogy: "beijing" is to "china" as "tokyo" is to what? The answer should be "japan".

```python
get_analogy('beijing', 'china', 'tokyo', glove_6b50d)
```

```
'japan'
```

"Adjective-superlative adjective" analogy: "bad" is to "worst" as "big" is to what? The answer should be "biggest".

```python
get_analogy('bad', 'worst', 'big', glove_6b50d)
```

```
'biggest'
```

"Present tense verb-past tense verb" analogy: "do" is to "did" as "go" is to what? The answer should be "went".

```python
get_analogy('do', 'did', 'go', glove_6b50d)
```

```
'went'
```

Summary

- Word vectors pre-trained on a large-scale corpus can often be applied to downstream natural language processing tasks.
- We can use pre-trained word vectors to seek synonyms and analogies.

Exercises

1. Test the fastText results.
2. If the dictionary is extremely large, how can we accelerate finding synonyms and analogies?

14.8 Text Classification and the Dataset

Text classification is a common task in natural language processing, which transforms a sequence of text of indefinite length into a category of text. It is similar to the image classification, the most frequently used application in this book, e.g., Section 17.8. The only difference is that, rather than an image, text classification's example is a text sentence.

This section will focus on loading data for one of the sub-questions in this field: using text sentiment classification to analyze the emotions of the text's author. This problem is also called sentiment analysis and has a wide range of applications. For example, we can analyze user reviews of products to obtain user satisfaction statistics, or analyze user sentiments about market conditions and use it to predict future trends.

```
import d2l
from mxnet import gluon, np, npx
import os
npx.set_np()
```

14.8.1 The Text Sentiment Classification Dataset

We use Stanford's Large Movie Review Dataset[226] as the dataset for text sentiment classification. This dataset is divided into two datasets for training and testing purposes, each containing 25,000 movie reviews downloaded from IMDb. In each dataset, the number of comments labeled as "positive" and "negative" is equal.

[226] https://ai.stanford.edu/~amaas/data/sentiment/

Reading the Dataset

We first download this dataset to the "../data" path and extract it to "../data/aclImdb".

```
# Saved in the d2l package for later use
d2l.DATA_HUB['aclImdb'] = (
    'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz',
    '01ada507287d82875905620988597833ad4e0903')

data_dir = d2l.download_extract('aclImdb', 'aclImdb')
```

```
Downloading ../data/aclImdb_v1.tar.gz from http://ai.stanford.edu/~amaas/data/sentiment/
↪aclImdb_v1.tar.gz...
```

Next, read the training and test datasets. Each example is a review and its corresponding label: 1 indicates "positive" and 0 indicates "negative".

```
# Saved in the d2l package for later use
def read_imdb(data_dir, is_train):
    data, labels = [], []
    for label in ['pos/', 'neg/']:
        folder_name = data_dir + ('train/' if is_train else 'test/') + label
        for file in os.listdir(folder_name):
            with open(folder_name+file, 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '')
                data.append(review)
                labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb(data_dir, is_train=True)
print('# trainings:', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[0:60])
```

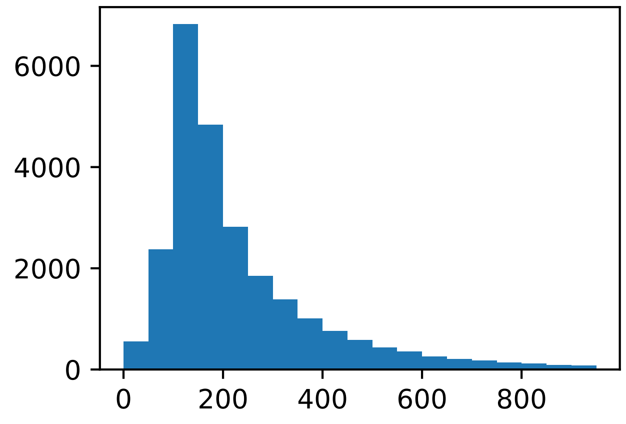
```
# trainings: 25000
label: 0 review: Normally the best way to annoy me in a film is to include so
label: 0 review: The Bible teaches us that the love of money is the root of a
label: 0 review: Being someone who lists Night of the Living Dead at number t
```

Tokenization and Vocabulary

We use a word as a token, and then create a dictionary based on the training dataset.

```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5)

d2l.set_figsize((3.5, 2.5))
d2l.plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 50));
```



Padding to the Same Length

Because the reviews have different lengths, so they cannot be directly combined into minibatches. Here we fix the length of each comment to 500 by truncating or adding "<unk>" indices.

```
num_steps = 500  # sequence length
train_features = np.array([d2l.trim_pad(vocab[line], num_steps, vocab.unk)
                           for line in train_tokens])
train_features.shape
```

```
(25000, 500)
```

Creating the Data Iterator

Now, we will create a data iterator. Each iteration will return a minibatch of data.

```
train_iter = d2l.load_array((train_features, train_data[1]), 64)

for X, y in train_iter:
    print('X', X.shape, 'y', y.shape)
    break
'# batches:', len(train_iter)
```

```
X (64, 500) y (64,)
```

```
('# batches:', 391)
```

14.8.2 Putting All Things Together

Last, we will save a function `load_data_imdb` into `d2l`, which returns the vocabulary and data iterators.

```python
# Saved in the d2l package for later use
def load_data_imdb(batch_size, num_steps=500):
    data_dir = d2l.download_extract('aclImdb', 'aclImdb')
    train_data = read_imdb(data_dir, True)
    test_data = read_imdb(data_dir, False)
    train_tokens = d2l.tokenize(train_data[0], token='word')
    test_tokens = d2l.tokenize(test_data[0], token='word')
    vocab = d2l.Vocab(train_tokens, min_freq=5)
    train_features = np.array([d2l.trim_pad(vocab[line], num_steps, vocab.unk)
                               for line in train_tokens])
    test_features = np.array([d2l.trim_pad(vocab[line], num_steps, vocab.unk)
                              for line in test_tokens])
    train_iter = d2l.load_array((train_features, train_data[1]), batch_size)
    test_iter = d2l.load_array((test_features, test_data[1]), batch_size,
                               is_train=False)
    return train_iter, test_iter, vocab
```

Summary

- Text classification can classify a text sequence into a category.
- To classify a text sentiment, we load an IMDb dataset and tokenize its words. Then we pad the text sequence for short reviews and create a data iterator.

Exercises

1. Discover a different natural language dataset (such as Amazon reviews[227]) and build a similar data_loader function as `load_data_imdb`.

14.9 Text Sentiment Classification: Using Recurrent Neural Networks

Similar to search synonyms and analogies, text classification is also a downstream application of word embedding. In this section, we will apply pre-trained word vectors and bidirectional recurrent neural networks with multiple hidden layers (Maas et al., 2011). We will use them to determine whether a text sequence of indefinite length contains positive or negative emotion. Import the required package or module before starting the experiment.

[227] https://snap.stanford.edu/data/web-Amazon.html

```python
import d2l
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn, rnn
from mxnet.contrib import text
npx.set_np()

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

14.9.1 Using a Recurrent Neural Network Model

In this model, each word first obtains a feature vector from the embedding layer. Then, we further encode the feature sequence using a bidirectional recurrent neural network to obtain sequence information. Finally, we transform the encoded sequence information to output through the fully connected layer. Specifically, we can concatenate hidden states of bidirectional long-short term memory in the initial timestep and final timestep and pass it to the output layer classification as encoded feature sequence information. In the `BiRNN` class implemented below, the `Embedding` instance is the embedding layer, the `LSTM` instance is the hidden layer for sequence encoding, and the `Dense` instance is the output layer for generated classification results.

```python
class BiRNN(nn.Block):
    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, **kwargs):
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Set Bidirectional to True to get a bidirectional recurrent neural
        # network
        self.encoder = rnn.LSTM(num_hiddens, num_layers=num_layers,
                                bidirectional=True, input_size=embed_size)
        self.decoder = nn.Dense(2)

    def forward(self, inputs):
        # The shape of inputs is (batch size, number of words). Because LSTM
        # needs to use sequence as the first dimension, the input is
        # transformed and the word feature is then extracted. The output shape
        # is (number of words, batch size, word vector dimension).
        embeddings = self.embedding(inputs.T)
        # Since the input (embeddings) is the only argument passed into
        # rnn.LSTM, it only returns the hidden states of the last hidden layer
        # at different timestep (outputs). The shape of outputs is
        # (number of words, batch size, 2 * number of hidden units).
        outputs = self.encoder(embeddings)
        # Concatenate the hidden states of the initial timestep and final
        # timestep to use as the input of the fully connected layer. Its
        # shape is (batch size, 4 * number of hidden units)
        encoding = np.concatenate((outputs[0], outputs[-1]), axis=1)
        outs = self.decoder(encoding)
        return outs
```

Create a bidirectional recurrent neural network with two hidden layers.

```
embed_size, num_hiddens, num_layers, ctx = 100, 100, 2, d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)
net.initialize(init.Xavier(), ctx=ctx)
```

Loading Pre-trained Word Vectors

Because the training dataset for sentiment classification is not very large, in order to deal with overfitting, we will directly use word vectors pre-trained on a larger corpus as the feature vectors of all words. Here, we load a 100-dimensional GloVe word vector for each word in the dictionary vocab.

```
glove_embedding = text.embedding.create(
    'glove', pretrained_file_name='glove.6B.100d.txt')
```

Query the word vectors that in our vocabulary.

```
embeds = glove_embedding.get_vecs_by_tokens(vocab.idx_to_token)
embeds.shape
```

```
(49339, 100)
```

Then, we will use these word vectors as feature vectors for each word in the reviews. Note that the dimensions of the pre-trained word vectors need to be consistent with the embedding layer output size `embed_size` in the created model. In addition, we no longer update these word vectors during training.

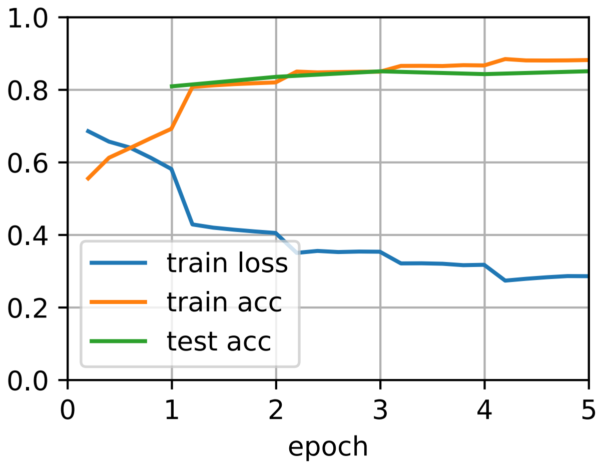
```
net.embedding.weight.set_data(embeds)
net.embedding.collect_params().setattr('grad_req', 'null')
```

Training and Evaluating the Model

Now, we can start training.

```
lr, num_epochs = 0.01, 5
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, ctx)
```

```
loss 0.286, train acc 0.882, test acc 0.851
644.1 exampes/sec on [gpu(0), gpu(1)]
```



Finally, define the prediction function.

```
# Saved in the d2l package for later use
def predict_sentiment(net, vocab, sentence):
    sentence = np.array(vocab[sentence.split()], ctx=d2l.try_gpu())
    label = np.argmax(net(sentence.reshape(1, -1)), axis=1)
    return 'positive' if label == 1 else 'negative'
```

Then, use the trained model to classify the sentiments of two simple sentences.

```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

Summary

- Text classification transforms a sequence of text of indefinite length into a category of text. This is a downstream application of word embedding.
- We can apply pre-trained word vectors and recurrent neural networks to classify the emotions in a text.

Exercises

1. Increase the number of epochs. What accuracy rate can you achieve on the training and testing datasets? What about trying to re-tune other hyper-parameters?
2. Will using larger pre-trained word vectors, such as 300-dimensional GloVe word vectors, improve classification accuracy?
3. Can we improve the classification accuracy by using the spaCy word tokenization tool? You need to install spaCy: `pip install spacy` and install the English package: `python -m spacy download en`. In the code, first import spacy: `import spacy`. Then, load the spacy English package: `spacy_en = spacy.load('en')`. Finally, define the function `def tokenizer(text): return [tok.text for tok in spacy_en.tokenizer(text)]` and replace the original `tokenizer` function. It should be noted that GloVe's word vector uses "-" to connect each word when storing noun phrases. For example, the phrase "new york" is represented as "new-york" in GloVe. After using spaCy tokenization, "new york" may be stored as "new york".

14.10 Text Sentiment Classification: Using Convolutional Neural Networks (textCNN)

In Chapter 6, we explored how to process two-dimensional image data with two-dimensional convolutional neural networks. In the previous language models and text classification tasks, we treated text data as a time series with only one dimension, and naturally, we used recurrent neural networks to process such data. In fact, we can also treat text as a one-dimensional image, so that we can use one-dimensional convolutional neural networks to capture associations between adjacent words. This section describes a groundbreaking approach to applying convolutional neural networks to text analysis: textCNN (Kim, 2014). First, import the packages and modules required for the experiment.

```python
import d2l
from mxnet import gluon, init, np, npx
from mxnet.contrib import text
from mxnet.gluon import nn
npx.set_np()

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

```
Downloading ../data/aclImdb_v1.tar.gz from http://ai.stanford.edu/~amaas/data/sentiment/
↪aclImdb_v1.tar.gz...
```

14.10.1 One-Dimensional Convolutional Layer

Before introducing the model, let's explain how a one-dimensional convolutional layer works. Like a two-dimensional convolutional layer, a one-dimensional convolutional layer uses a one-dimensional cross-correlation operation. In the one-dimensional cross-correlation operation, the convolution window starts from the leftmost side of the input array and slides on the input array from left to right successively. When the convolution window slides to a certain position, the input subarray in the window and kernel array are multiplied and summed by element to get the element at the corresponding location in the output array. As shown in Fig. 14.10.1, the input is a one-dimensional array with a width of 7 and the width of the kernel array is 2. As we can see, the output width is $7 - 2 + 1 = 6$ and the first element is obtained by performing multiplication by element on the leftmost input subarray with a width of 2 and kernel array and then summing the results.

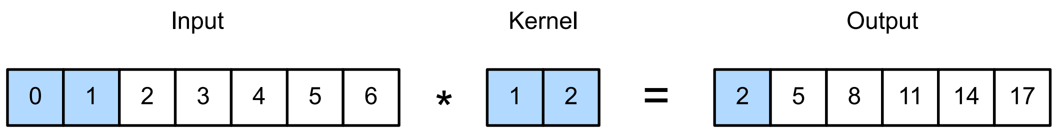


Fig. 14.10.1: One-dimensional cross-correlation operation. The shaded parts are the first output element as well as the input and kernel array elements used in its calculation: $0 \times 1 + 1 \times 2 = 2$.

Next, we implement one-dimensional cross-correlation in the `corr1d` function. It accepts the input array `X` and kernel array `K` and outputs the array `Y`.

```python
def corr1d(X, K):
    w = K.shape[0]
    Y = np.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i: i + w] * K).sum()
    return Y
```

Now, we will reproduce the results of the one-dimensional cross-correlation operation in Fig. 14.10.1.

```python
X, K = np.array([0, 1, 2, 3, 4, 5, 6]), np.array([1, 2])
corr1d(X, K)
```

```
array([ 2.,  5.,  8., 11., 14., 17.])
```

The one-dimensional cross-correlation operation for multiple input channels is also similar to the two-dimensional cross-correlation operation for multiple input channels. On each channel, it performs the one-dimensional cross-correlation operation on the kernel and its corresponding input and adds the results of the channels to get the output. Fig. 14.10.2 shows a one-dimensional cross-correlation operation with three input channels.

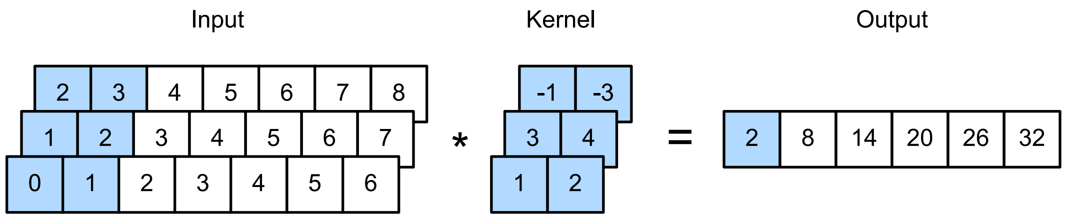


Fig. 14.10.2: One-dimensional cross-correlation operation with three input channels. The shaded parts are the first output element as well as the input and kernel array elements used in its calculation: $0 \times 1 + 1 \times 2 + 1 \times 3 + 2 \times 4 + 2 \times (-1) + 3 \times (-3) = 2$.

Now, we reproduce the results of the one-dimensional cross-correlation operation with multi-input channel in Fig. 14.10.2.

```
def corr1d_multi_in(X, K):
    # First, we traverse along the 0th dimension (channel dimension) of X and
    # K. Then, we add them together by using * to turn the result list into a
    # positional argument of the add_n function
    return sum(corr1d(x, k) for x, k in zip(X, K))

X = np.array([[0, 1, 2, 3, 4, 5, 6],
              [1, 2, 3, 4, 5, 6, 7],
              [2, 3, 4, 5, 6, 7, 8]])
K = np.array([[1, 2], [3, 4], [-1, -3]])
corr1d_multi_in(X, K)
```

```
array([ 2.,  8., 14., 20., 26., 32.])
```

The definition of a two-dimensional cross-correlation operation tells us that a one-dimensional cross-correlation operation with multiple input channels can be regarded as a two-dimensional cross-correlation operation with a single input channel. As shown in Fig. 14.10.3, we can also present the one-dimensional cross-correlation operation with multiple input channels in Fig. 14.10.2 as the equivalent two-dimensional cross-correlation operation with a single input channel. Here, the height of the kernel is equal to the height of the input.

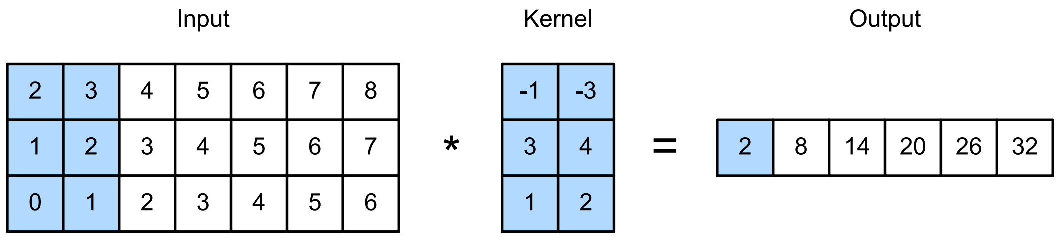


Fig. 14.10.3: Two-dimensional cross-correlation operation with a single input channel. The highlighted parts are the first output element and the input and kernel array elements used in its calculation: $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$.

Both the outputs in Fig. 14.10.1 and Fig. 14.10.2 have only one channel. We discussed how to specify multiple output channels in a two-dimensional convolutional layer in Section 6.4. Similarly, we can also specify multiple output channels in the one-dimensional convolutional layer to extend the model parameters in the convolutional layer.

14.10.2 Max-Over-Time Pooling Layer

Similarly, we have a one-dimensional pooling layer. The max-over-time pooling layer used in TextCNN actually corresponds to a one-dimensional global maximum pooling layer. Assuming that the input contains multiple channels, and each channel consists of values on different timesteps, the output of each channel will be the largest value of all timesteps in the channel. Therefore, the input of the max-over-time pooling layer can have different timesteps on each channel.

To improve computing performance, we often combine timing examples of different lengths into a minibatch and make the lengths of each timing example in the batch consistent by appending special characters (such as 0) to the end of shorter examples. Naturally, the added special characters have no intrinsic meaning. Because the main purpose of the max-over-time pooling layer is to capture the most important features of timing, it usually allows the model to be unaffected by the manually added characters.

14.10.3 The TextCNN Model

TextCNN mainly uses a one-dimensional convolutional layer and max-over-time pooling layer. Suppose the input text sequence consists of n words, and each word is represented by a d-dimension word vector. Then the input example has a width of n, a height of 1, and d input channels. The calculation of textCNN can be mainly divided into the following steps:

1. Define multiple one-dimensional convolution kernels and use them to perform convolution calculations on the inputs. Convolution kernels with different widths may capture the correlation of different numbers of adjacent words.
2. Perform max-over-time pooling on all output channels, and then concatenate the pooling output values of these channels in a vector.
3. The concatenated vector is transformed into the output for each category through the fully connected layer. A dropout layer can be used in this step to deal with overfitting.

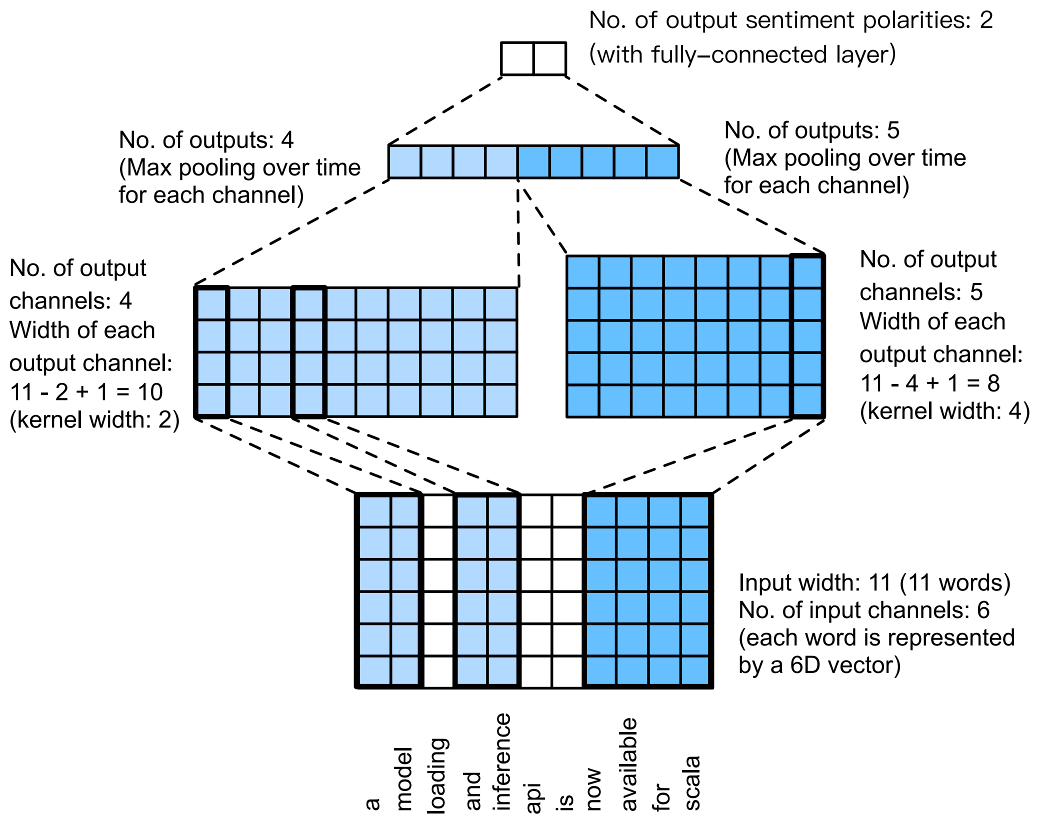


Fig. 14.10.4: TextCNN design.

Fig. 14.10.4 gives an example to illustrate the textCNN. The input here is a sentence with 11 words, with each word represented by a 6-dimensional word vector. Therefore, the input sequence has a width of 11 and 6 input channels. We assume there are two one-dimensional convolution kernels with widths of 2 and 4, and 4 and 5 output channels, respectively. Therefore, after one-dimensional convolution calculation, the width of the four output channels is $11 - 2 + 1 = 10$, while the width of the other five channels is $11 - 4 + 1 = 8$. Even though the width of each channel is different, we can still perform max-over-time pooling for each channel and concatenate the pooling outputs of the 9 channels into a 9-dimensional vector. Finally, we use a fully connected layer to transform the 9-dimensional vector into a 2-dimensional output: positive sentiment and negative sentiment predictions.

Next, we will implement a textCNN model. Compared with the previous section, in addition to replacing the recurrent neural network with a one-dimensional convolutional layer, here we use two embedding layers, one with a fixed weight and another that participates in training.

```
class TextCNN(nn.Block):
    def __init__(self, vocab_size, embed_size, kernel_sizes, num_channels,
                 **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # The embedding layer does not participate in training
        self.constant_embedding = nn.Embedding(vocab_size, embed_size)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Dense(2)
        # The max-over-time pooling layer has no weight, so it can share an
```

(continues on next page)

(continued from previous page)

```
        # instance
        self.pool = nn.GlobalMaxPool1D()
        # Create multiple one-dimensional convolutional layers
        self.convs = nn.Sequential()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.add(nn.Conv1D(c, k, activation='relu'))

    def forward(self, inputs):
        # Concatenate the output of two embedding layers with shape of
        # (batch size, number of words, word vector dimension) by word vector
        embeddings = np.concatenate((
            self.embedding(inputs), self.constant_embedding(inputs)), axis=2)
        # According to the input format required by Conv1D, the word vector
        # dimension, that is, the channel dimension of the one-dimensional
        # convolutional layer, is transformed into the previous dimension
        embeddings = embeddings.transpose(0, 2, 1)
        # For each one-dimensional convolutional layer, after max-over-time
        # pooling, an ndarray with the shape of (batch size, channel size, 1)
        # can be obtained. Use the flatten function to remove the last
        # dimension and then concatenate on the channel dimension
        encoding = np.concatenate([
            np.squeeze(self.pool(conv(embeddings)), axis=-1)
            for conv in self.convs], axis=1)
        # After applying the dropout method, use a fully connected layer to
        # obtain the output
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

Create a TextCNN instance. It has 3 convolutional layers with kernel widths of 3, 4, and 5, all with 100 output channels.

```
embed_size, kernel_sizes, nums_channels = 100, [3, 4, 5], [100, 100, 100]
ctx = d2l.try_all_gpus()
net = TextCNN(len(vocab), embed_size, kernel_sizes, nums_channels)
net.initialize(init.Xavier(), ctx=ctx)
```

Load Pre-trained Word Vectors

As in the previous section, load pre-trained 100-dimensional GloVe word vectors and initialize the embedding layers `embedding` and `constant_embedding`. Here, the former participates in training while the latter has a fixed weight.

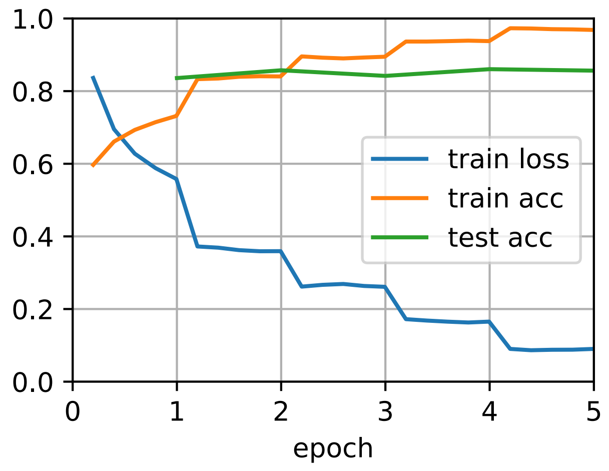
```
glove_embedding = text.embedding.create(
    'glove', pretrained_file_name='glove.6B.100d.txt')
embeds = glove_embedding.get_vecs_by_tokens(vocab.idx_to_token)
net.embedding.weight.set_data(embeds)
net.constant_embedding.weight.set_data(embeds)
net.constant_embedding.collect_params().setattr('grad_req', 'null')
```

Train and Evaluate the Model

Now we can train the model.

```
lr, num_epochs = 0.001, 5
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, ctx)
```

```
loss 0.090, train acc 0.968, test acc 0.856
4205.6 exampes/sec on [gpu(0), gpu(1)]
```



Below, we use the trained model to classify sentiments of two simple sentences.

```
d2l.predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
d2l.predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

Summary

- We can use one-dimensional convolution to process and analyze timing data.
- A one-dimensional cross-correlation operation with multiple input channels can be regarded as a two-dimensional cross-correlation operation with a single input channel.
- The input of the max-over-time pooling layer can have different numbers of timesteps on each channel.
- TextCNN mainly uses a one-dimensional convolutional layer and max-over-time pooling layer.

Exercises

1. Tune the hyper-parameters and compare the two sentiment analysis methods, using recurrent neural networks and using convolutional neural networks, as regards accuracy and operational efficiency.
2. Can you further improve the accuracy of the model on the test set by using the three methods introduced in the previous section: tuning hyper-parameters, using larger pre-trained word vectors, and using the spaCy word tokenization tool?
3. What other natural language processing tasks can you use textCNN for?