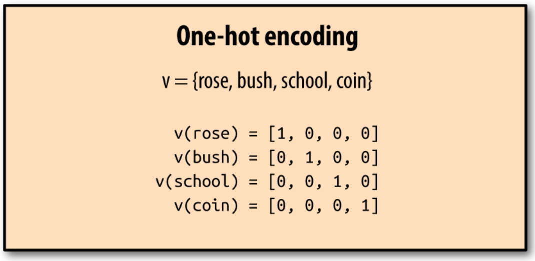
entries of this vector are zero, except for one entry, at the index that corresponds to the current word. For an example of this embedding, see Figure 7-10.



One-hot encoding

v = {rose, bush, school, coin}

```
  v(rose) = [1, 0, 0, 0]
  v(bush) = [0, 1, 0, 0]
v(school) = [0, 0, 1, 0]
  v(coin) = [0, 0, 0, 1]
```

Figure 7-10. One-hot encodings transform words into vectors with only one nonzero entry (which is typically set to one). Different indices in the vector uniquely represent words in a language corpus.

It's also possible to use more sophisticated embeddings. The basic idea is similar to that for the one-hot encoding. Each word is associated with a unique vector. However, the key difference is that it's possible to learn this encoding vector directly from data to obtain a "word embedding" for the word in question that's meaningful for the dataset at hand. We will show you how to learn word embeddings later in this chapter.

In order to process the Penn Treebank data, we need to find the vocabulary of words used in the corpus, then transform each word into its associated word vector. We will then show how to feed the processed data into a TensorFlow model.

Penn Treebank Limitations

The Penn Treebank is a very useful dataset for language modeling, but it no longer poses a challenge for state-of-the-art language models; researchers have already overfit models on the peculiarities of this collection. State-of-the-art research would use larger datasets such as the billion-word-corpus language benchmark. However, for our exploratory purposes, the Penn Treebank easily suffices.

Code for Preprocessing

The snippet of code in Example 7-1 reads in the raw files associated with the Penn Treebank corpus. The corpus is stored with one sentence per line. Some Python string handling is done to replace `"\n"` newline markers with fixed-token `"<eos>"` and then split the file into a list of tokens.

Example 7-1. This function reads in the raw Penn Treebank file

```
def _read_words(filename):
  with tf.gfile.GFile(filename, "r") as f:
    if sys.version_info[0] >= 3:
      return f.read().replace("\n", "<eos>").split()
    else:
      return f.read().decode("utf-8").replace("\n", "<eos>").split()
```

With `_read_words` defined, we can build the vocabulary associated with a given file using function `_build_vocab` defined in Example 7-2. We simply read in the words in the file, and count the number of unique words in the file using Python's `collections` library. For convenience, we construct a dictionary object mapping words to their unique integer identifiers (their positions in the vocabulary). Tying it all together, `_file_to_word_ids` transforms a file into a list of word identifiers (Example 7-3).

Example 7-2. This function builds a vocabulary consisting of all words in the specified file

```
def _build_vocab(filename):
  data = _read_words(filename)

  counter = collections.Counter(data)
  count_pairs = sorted(counter.items(), key=lambda x: (-x[1], x[0]))

  words, _ = list(zip(*count_pairs))
  word_to_id = dict(zip(words, range(len(words))))

  return word_to_id
```

Example 7-3. This function transforms words in a file into id numbers

```
def _file_to_word_ids(filename, word_to_id):
  data = _read_words(filename)
  return [word_to_id[word] for word in data if word in word_to_id]
```

With these utilities in place, we can process the Penn Treebank corpus with function `ptb_raw_data` (Example 7-4). Note that training, validation, and test datasets are prespecified, so we need only read each file into a list of unique indices.

Example 7-4. This function loads the Penn Treebank data from the specified location

```
def ptb_raw_data(data_path=None):
  """Load PTB raw data from data directory "data_path".

  Reads PTB text files, converts strings to integer ids,
  and performs mini-batching of the inputs.

  The PTB dataset comes from Tomas Mikolov's webpage:
  http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz

  Args:
    data_path: string path to the directory where simple-examples.tgz
               has been extracted.

  Returns:
    tuple (train_data, valid_data, test_data, vocabulary)
    where each of the data objects can be passed to PTBIterator.
  """

  train_path = os.path.join(data_path, "ptb.train.txt")
  valid_path = os.path.join(data_path, "ptb.valid.txt")
  test_path = os.path.join(data_path, "ptb.test.txt")

  word_to_id = _build_vocab(train_path)
  train_data = _file_to_word_ids(train_path, word_to_id)
  valid_data = _file_to_word_ids(valid_path, word_to_id)
  test_data = _file_to_word_ids(test_path, word_to_id)
  vocabulary = len(word_to_id)
  return train_data, valid_data, test_data, vocabulary
```

tf.GFile and tf.Flags

TensorFlow is a large project that contains many bits and pieces. While most of the library is devoted to machine learning, there's also a large proportion that's dedicated to loading and massaging data. Some of these functions provide useful capabilities that aren't found elsewhere. Other parts of the loading functionality are less useful, however.

`tf.GFile` and `tf.FLags` provide functionality that is more or less identical to standard Python file handling and `argparse`. The provenance of these tools is historical. With Google, custom file handlers and flag handling are required by internal code standards. For the rest of us, though, it's better style to use standard Python tools whenever possible. It's much better for readability and stability.

Loading Data into TensorFlow

In this section, we cover the code needed to load our processed indices into TensorFlow. To do so, we will introduce you to a new bit of TensorFlow machinery. Until now, we've used feed dictionaries to pass data into TensorFlow. While feed dictionaries are fine for small toy datasets, they are often not good choices for larger datasets, since large Python overheads involving packing and unpacking dictionaries are introduced. For more performant code, it's better to use TensorFlow queues.