Recurrent Networks Are Turing Complete

Perhaps unsurprisingly, NTMs are capable of performing any computation a Turing machine can and are consequently Turing complete. However, a less known fact is that vanilla recurrent neural networks are themselves Turing complete! Put another way, in principle, a recurrent neural network is capable of learning to perform arbitrary computation.

The basic idea is that the transition operator can learn to perform basic reading, writing, and storage operations. The unrolling of the recurrent network over time allows for the performance of complex computations. In some sense, this fact shouldn't be too surprising. The universal approximation theorem already demonstrates that fully connected networks are capable of learning arbitrary functions. Chaining arbitrary functions together over time leads to arbitrary computations. (The technical details required to formally prove this are formidable, though.)

Working with Recurrent Neural Networks in Practice

In this section, you will learn about the use of recurrent neural networks for language modeling on the Penn Treebank dataset, a natural language dataset built from *Wall Street Journal* articles. We will introduce the TensorFlow primitives needed to perform this modeling and will also walk you through the data handling and preprocessing steps needed to prepare data for training. We encourage you to follow along and try running the code in the GitHub repo associated with the book.

Processing the Penn Treebank Corpus

The Penn Treebank contains a million-word corpus of *Wall Street Journal* articles. This corpus can be used for either character-level or word-level modeling (the tasks of predicting the next character or word in a sentence given those preceding). The efficacy of models is measured using the perplexity of trained models (more on this metric later).

The Penn Treebank corpus consists of sentences. How can we transform sentences into a form that can be fed to machine learning systems such as recurrent language models? Recall that machine learning models accept tensors (with recurrent models accepting sequences of tensors) as input. Consequently, we need to transform words into tensors for machine learning.

The simplest method of transforming words into vectors is to use "one-hot" encoding. In this encoding, let's suppose that our language dataset uses a vocabulary that has $|V|$ words. Then each word is transformed into a vector of shape $(|V|)$. All the

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.

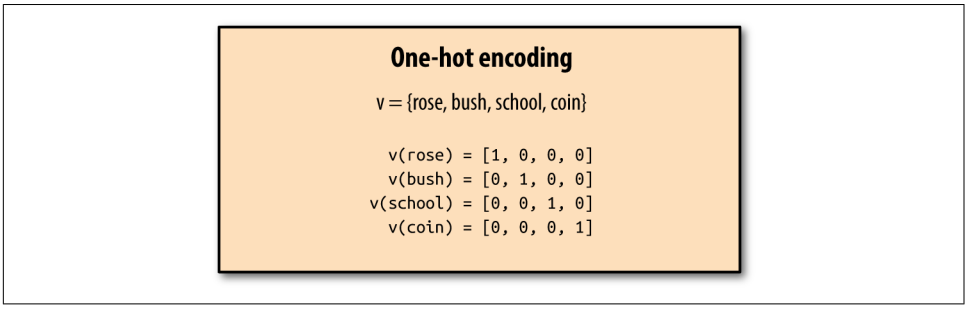

Figure 7-10. One-hot encodings transform words into vectors with only one nonzero entry (which is typically set to one). Different indices in the vector uniquely represent words in a language corpus.

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.