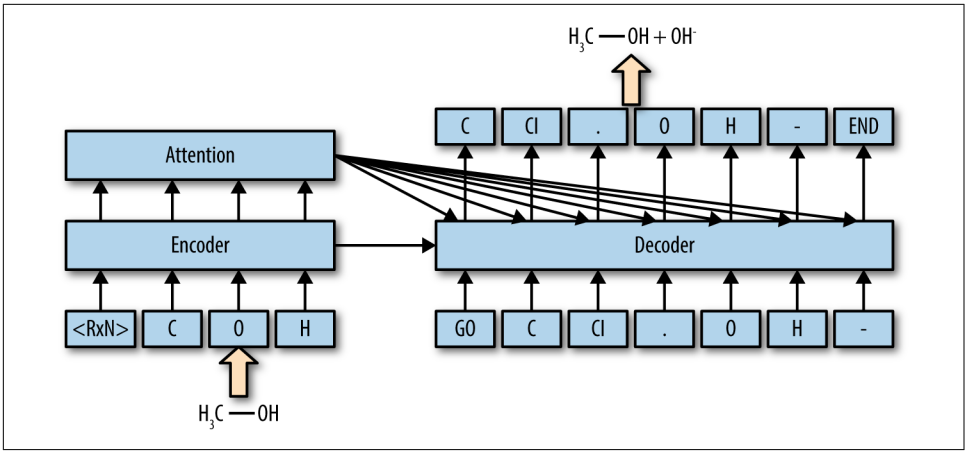

Figure 7-8. A seq2seq model for chemical retrosynthesis transforms a sequence of chemical products into a sequence of chemical reactants.

Neural Turing Machines

The dream of machine learning has been to move further up the abstraction stack: moving from learning short pattern-matching engines to learning to perform arbitrary computations. The Neural Turing machine is a powerful step in this evolution.

The Turing machine was a seminal contribution to the mathematical theory of computation. It was the first mathematical model of a machine capable of performing any computation. The Turing machine maintains a "tape" that provides a memory of the performed computation. The second part of the machine is a "head" that performs transformations on single tape cells. The insight of the Turing machine was that the "head" didn't need to be very complicated in order to perform arbitrarily complicated calculations.

The Neural Turing machine (NTM) is a very clever attempt to transmute a Turing machine itself into a neural network. The trick in this transmutation is to turn discrete actions into soft continuous functions (this is a trick that pops up in deep learning repeatedly, so take note!)

The Turing machine head is quite similar to the RNN cell! As a result, the NTM can be trained end-to-end to learn to perform arbitrary computations, in principle at least (Figure 7-9). In practice, there are severe limitations to the set of computations that the NTM can perform. Gradient flow instabilities (as always) limit what can be learned. More research and experimentation will be needed to devise successors to NTMs capable of learning more useful functions.

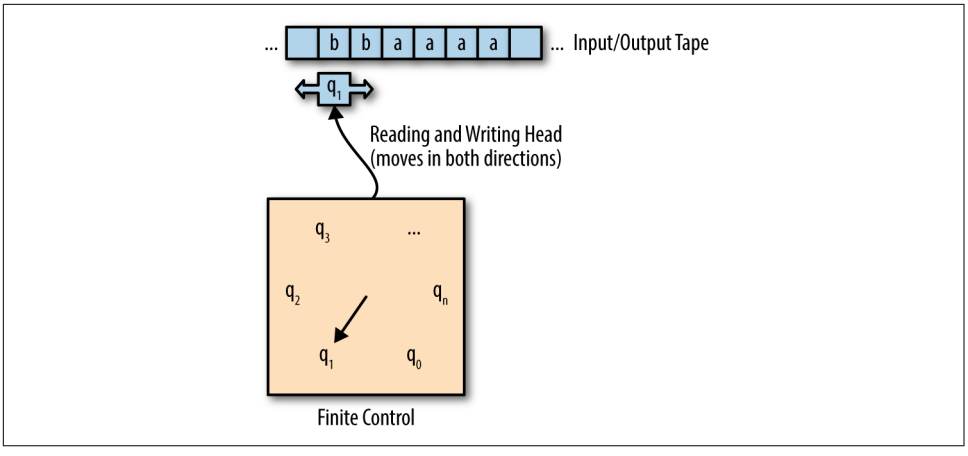

Figure 7-9. A Neural Turing machine (NTM) is a learnable version of a Turing machine. It maintains a tape where it can store the outputs of intermediate computations. While NTMs have many practical limitations, it's possible that their intellectual descendants will be capable of learning powerful algorithms.

Turing Completeness

The concept of Turing completeness is an important notion in computer science. A programming language is said to be Turing complete if it is capable of performing any computation that can be performed by a Turing machine. The Turing machine itself was invented to provide a mathematical model of what it means for a function to be "computable." The machine provides the capability to read, write, and store in memory various instructions, abstract primitives that underlie all computing machines.

Over time, a large body of work has shown that the Turing machine closely models the set of computations performable in the physical world. To a first approximation, if it can be shown that a Turing machine is incapable of performing a computation, no computing device is capable of it either. On the other side, if it can be shown that a computing system can perform the basic operations of a Turing machine, it is then "Turing complete" and capable of performing in principle any computation that can be performed at all. A number of surprising systems are Turing complete. We encourage you to read more about this topic if interested.

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