CHAPTER

10 Encoder-Decoder Models, Attention, and Contextual Embeddings

It is all well and good to copy what one sees, but it is much better to draw only what remains in one's memory. This is a transformation in which imagination and memory collaborate.

Edgar Degas

In Chapter 9 we explored recurrent neural networks along with some of their common use cases, including language modeling, contextual generation, and sequence labeling. A common thread in these applications was the notion of transduction — input sequences being transformed into output sequences in a one-to-one fashion. Here, we'll explore an approach that extends these models and provides much greater flexibility across a range of applications. Specifically, we'll introduce **encoder-decoder** networks, or **sequence-to-sequence** models, that are capable of generating contextually appropriate, arbitrary length, output sequences. Encoder-decoder networks have been applied to a very wide range of applications including machine translation, summarization, question answering, and dialogue modeling.

The key idea underlying these networks is the use of an **encoder** network that takes an input sequence and creates a contextualized representation of it. This representation is then passed to a **decoder** which generates a task-specific output sequence. The encoder and decoder networks are typically implemented with the same architecture, often using recurrent networks of the kind we studied in Chapter 9. And as with the deep networks introduced there, the encoder-decoder architecture allows networks to be trained in an end-to-end fashion for each application.

10.1 Neural Language Models and Generation Revisited

To understand the design of encoder-decoder networks let's return to neural language models and the notion of autoregressive generation. Recall that in a simple recurrent network, the value f the hidden state at a particular point in time is a function of the previous hidden state and the current input; the network output is then a function of this new hidden state.

$$\begin{aligned} h_t &= g(Uh_{t-1} + Wx_t) \\ y_t &= f(Vh_t) \end{aligned}$$

Here, U, V, and W are weight matrices which are adjusted during training, g is a suitable non-linear activation function such as *tanh* or ReLU, and in the common case of classification f is a softmax over the set of possible outputs. In practice, gated networks using LSTMs or GRUs are used in place of these simple RNNs. To

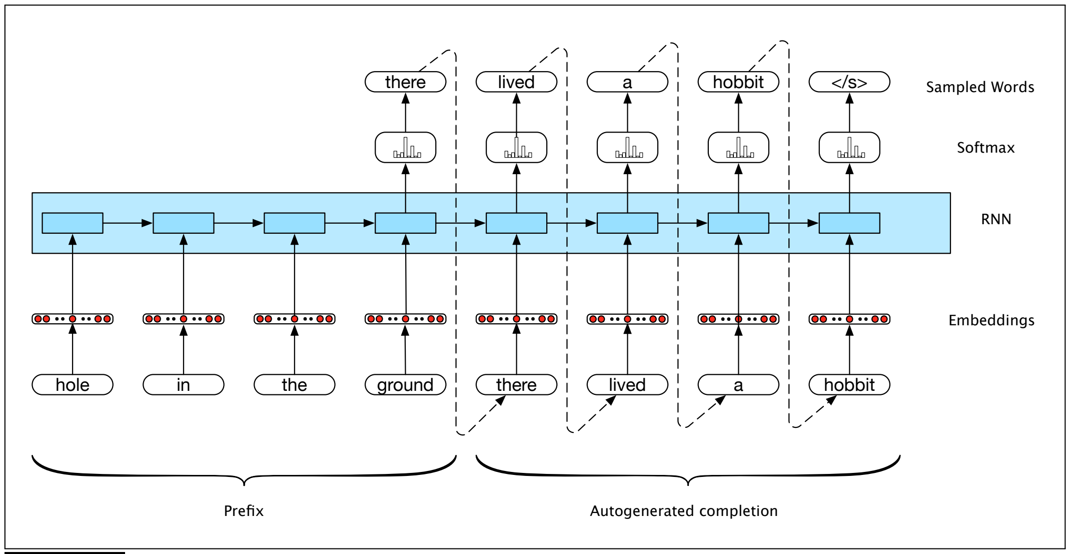

Figure 10.1 Using an RNN to generate the completion of an input phrase.

reflect this, we'll abstract away from the details of the specific RNN being used and simply specify the inputs on which the computation is being based. So the earlier equations will be expressed as follows with the understanding that there is a suitable RNN underneath.

$$\begin{aligned} h_t &= g(h_{t-1}, x_t) \\ y_t &= f(h_t) \end{aligned}$$

To create an RNN-based language model, we train the network to predict the next word in a sequence using a corpus of representative text. Language models trained in this fashion are referred to as autoregressive models. Given a trained model, we can ask the network to generate novel sequences by first randomly sampling an appropriate word as the beginning of a sequence. We then condition the generation of subsequent words on the hidden state from the previous time step as well as the embedding for the word just generated, again sampling from the distribution provided by the softmax. More specifically, during generation the softmax output at each point in time provides us with the probability of every word in the vocabulary given the preceding context, that is $P(y_i|y_{<i}) \forall i \in V$; we then sample a particular word, $\hat{y}_i$, from this distribution and condition subsequent generation on it. The process continues until the end of sentence token <\s> is generated.

Now, let's consider a simple variation on this scheme. Instead of having the language model generate a sentence from scratch, we have it complete a sequence given a specified prefix. More specifically, we first pass the specified prefix through the language model using forward inference to produce a sequence of hidden states, ending with the hidden state corresponding to the last word of the prefix. We then begin generating as we did earlier, but using the final hidden state of the prefix as our starting point. The result of this process is a novel output sequence that should be a reasonable completion of the prefix input.

Fig. 10.1 illustrates this basic scheme. The portion of the network on the left processes the provided prefix, while the right side executes the subsequent autoregressive generation. Note that the goal of the lefthand portion of the network is to generate a series of hidden states from the given input; there are no outputs associated with this part of the process until we reach the end of the prefix.

Now, consider an ingenious extension of this idea from the world of machine translation (MT), the task of automatically translating sentences from one language into another. The primary resources used to train modern translation systems are known as parallel texts, or **bitexts**. These are large text collections consisting of pairs of sentences from different languages that are translations of one another. Traditionally in MT, the text being translated is referred to as the **source** and the translation output is called the **target**.

bitexts

To extend language models and autoregressive generation to machine translation, we'll first add an end-of-sentence marker at the end of each bitext's source sentence and then simply concatenate the corresponding target to it. These concatenated source-target pairs can now serve as training data for a combined language model. Training proceeds as with any RNN-based language model. The network is trained autoregressively to predict the next word in a set of sequences comprised of the concatenated source-target bitexts, as shown in Fig. 10.2.

To translate a source text using the trained model, we run it through the network performing forward inference to generate hidden states until we get to the end of the source. Then we begin autoregressive generation, asking for a word in the context of the hidden layer from the end of the source input as well as the end-of-sentence marker. Subsequent words are conditioned on the previous hidden state and the embedding for the last word generated.

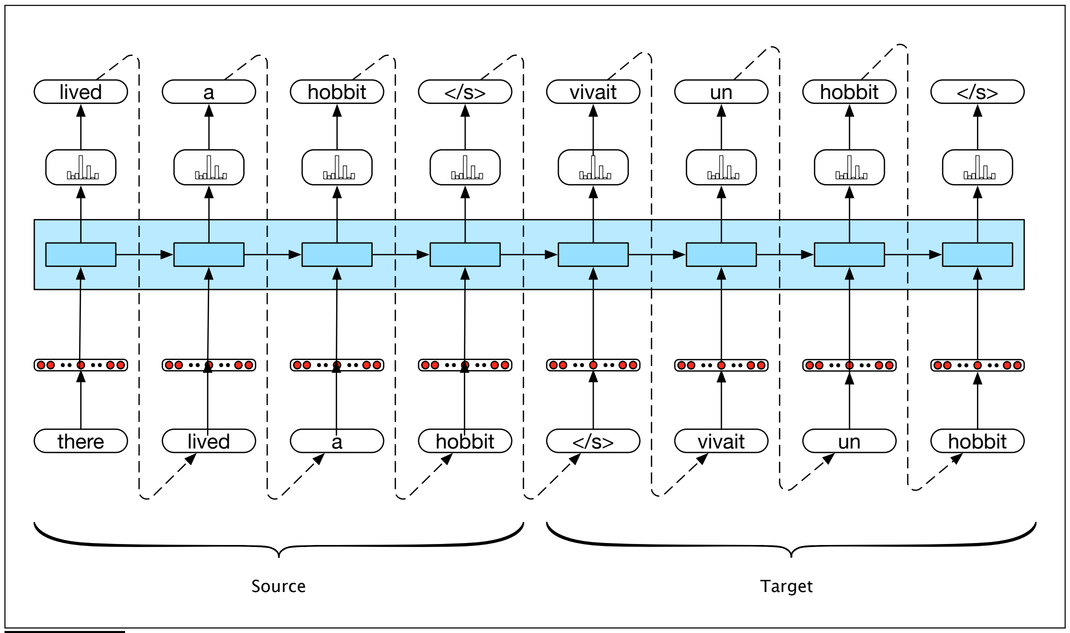

Figure 10.2 Training setup for a neural language model approach to machine translation. Source-target bitexts are concatenated and used to train a language model.

Early efforts using this clever approach demonstrated surprisingly good results on standard datasets and led to a series of innovations that were the basis for networks discussed in the remainder of this chapter. Chapter 11 provides an in-depth discussion of the fundamental issues in translation as well as the current state-of-the-art approaches to MT. Here, we'll focus on the powerful models that arose from these early efforts.

10.2 Encoder-Decoder Networks

encoder-decoder

Fig. 10.3 abstracts away from the specifics of machine translation and illustrates a basic **encoder-decoder** architecture. The elements of the network on the left process the input sequence and comprise the **encoder**, the entire purpose of which is to generate a contextualized representation of the input. In this network, this representation is embodied in the final hidden state of the encoder, h_n, which in turn feeds into the first hidden state of the decoder. The **decoder** network on the right takes this state and autoregressively generates a sequence of outputs.

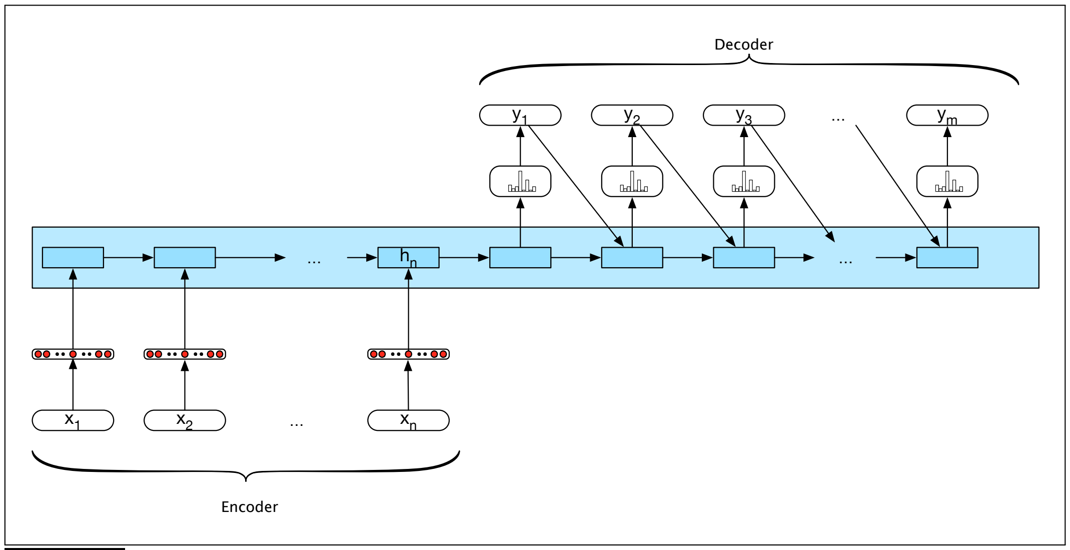

Figure 10.3 Basic RNN-based encoder-decoder architecture. The final hidden state of the encoder RNN serves as the context for the decoder in its role as h_0 in the decoder RNN.

This basic architecture is consistent with the original applications of neural models to machine translation. However, it embodies a number of design choices that are less than optimal. Among the major ones are that the encoder and the decoder are assumed to have the same internal structure (RNNs in this case), that the final state of the encoder is the only context available to the decoder, and finally that this context is only available to the decoder as its initial hidden state. Abstracting away from these choices, we can say that encoder-decoder networks consist of three components:

1. An **encoder** that accepts an input sequence, x_1^n, and generates a corresponding sequence of contextualized representations, h_1^n.
2. A **context vector**, c, which is a function of h_1^n, and conveys the essence of the input to the decoder.
3. And a **decoder**, which accepts c as input and generates an arbitrary length sequence of hidden states h_1^m, from which a corresponding sequence of output states y_1^m, can be obtained.

Fig. 10.4 illustrates this abstracted architecture. Let's now explore some of the possibilities for each of the components.

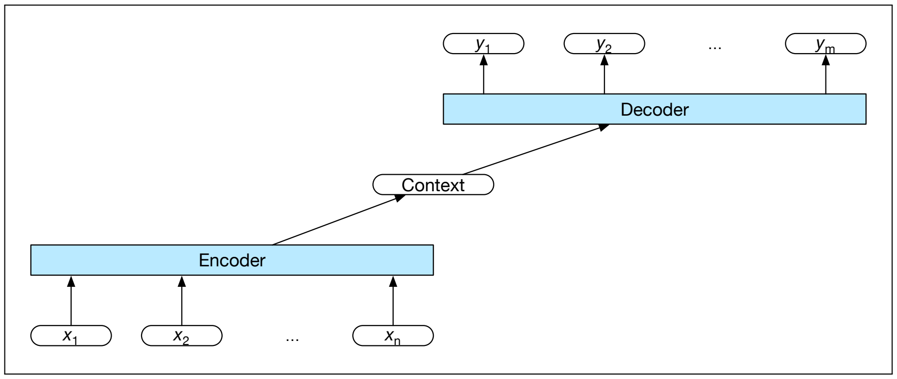

Figure 10.4 Basic architecture for an abstract encoder-decoder network. The context is a function of the vector of contextualized input representations and may be used by the decoder in a variety of ways.

Encoder

Simple RNNs, LSTMs, GRUs, convolutional networks, as well as transformer networks (discussed later in this chapter), can all be been employed as encoders. For simplicity, our figures show only a single network layer for the encoder, however, stacked architectures are the norm, where the output states from the top layer of the stack are taken as the final representation. A widely used encoder design makes use of stacked Bi-LSTMs where the hidden states from top layers from the forward and backward passes are concatenated as described in Chapter 9 to provide the contextualized representations for each time step.

Decoder

For the decoder, autoregressive generation is used to produce an output sequence, an element at a time, until an end-of-sequence marker is generated. This incremental process is guided by the context provided by the encoder as well as any items generated for earlier states by the decoder. Again, a typical approach is to use an LSTM or GRU-based RNN where the context consists of the final hidden state of the encoder, and is used to initialize the first hidden state of the decoder. (To help keep things straight, we'll use the superscripts *e* and *d* where needed to distinguish the hidden states of the encoder and the decoder.) Generation proceeds as described earlier where each hidden state is conditioned on the previous hidden state and output generated in the previous state.

$$\begin{aligned} c &= h_n^e \\ h_0^d &= c \\ \\ h_t^d &= g(\hat{y}_{t-1}, h_{t-1}^d) \\ z_t &= f(h_t^d) \\ y_t &= \text{softmax}(z_t) \end{aligned}$$

Recall, that g is a stand-in for some flavor of RNN and $\hat{y}_{t-1}$ is the embedding for the output sampled from the softmax at the previous step.

A weakness of this approach is that the context vector, c, is only directly available at the beginning of the process and its influence will wane as the output sequence is generated. A solution is to make the context vector c available at each step in the decoding process by adding it as a parameter to the computation of the current hidden state.

$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d, c)$$

A common approach to the calculation of the output layer y is to base it solely on this newly computed hidden state. While this cleanly separates the underlying recurrence from the output generation task, it makes it difficult to keep track of what has already been generated and what hasn't. A alternative approach is to condition the output on both the newly generated hidden state, the output generated at the previous state, and the encoder context.

$$y_t = \text{softmax}(\hat{y}_{t-1}, z_t, c)$$

Finally, as shown earlier, the output y at each time consists of a softmax computation over the set of possible outputs (the vocabulary in the case of language models). What one does with this distribution is task-dependent, but it is critical since the recurrence depends on choosing a particular output, $\hat{y}$, from the softmax to condition the next step in decoding. We've already seen several of the possible options for this. For neural generation, where we are trying to generate novel outputs, we can simply sample from the softmax distribution. However, for applications like MT where we're looking for a specific output sequence, random sampling isn't appropriate and would likely lead to some strange output. An alternative is to choose the most likely output at each time step by taking the argmax over the softmax output:

$$\hat{y} \;=\; \text{argmax} P(y_i | y_< i)$$

This is easy to implement but as we've seen several times with sequence labeling, independently choosing the argmax over a sequence is not a reliable way of arriving at a good output since it doesn't guarantee that the individual choices being made make sense together and combine into a coherent whole. With sequence labeling we addressed this with a CRF-layer over the output token types combined with a Viterbi-style dynamic programming search. Unfortunately, this approach is not viable here since the dynamic programming invariant doesn't hold.

Beam Search

A viable alternative is to view the decoding problem as a heuristic state-space search and systematically explore the space of possible outputs. The key to such an approach is controlling the exponential growth of the search space. To accomplish this, we'll use a technique called **beam search**. Beam search operates by combining a breadth-first-search strategy with a heuristic filter that scores each option and prunes the search space to stay within a fixed-size memory footprint, called the beam width.

Beam Search

At the first step of decoding, we select the B-best options from the softmax output y, where B is the size of the beam. Each option is scored with its corresponding probability from the softmax output of the decoder. These initial outputs constitute the search frontier. We'll refer to the sequence of partial outputs generated along these search paths as **hypotheses**.

At subsequent steps, each hypothesis on the frontier is extended incrementally by being passed to distinct decoders, which again generate a softmax over the entire vocabulary. To provide the necessary inputs for the decoders, each hypothesis must include not only the words generated thus far but also the context vector, and the hidden state from the previous step. New hypotheses representing every possible extension to the current ones are generated and added to the frontier. Each of these new hypotheses is scored using $P(y_i|y_{<i})$, which is the product of the probability of current word choice multiplied by the probability of the path that led to it. To control the exponential growth of the frontier, it is pruned to contain only the top B hypotheses.

This process continues until a <\s> is generated indicating that a complete candidate output has been found. At this point, the completed hypothesis is removed from the frontier and the size of the beam is reduced by one. The search continues until the beam has been reduced to 0. Leaving us with B hypotheses to consider. Fig. 10.5 illustrates this process with a beam width of 3.

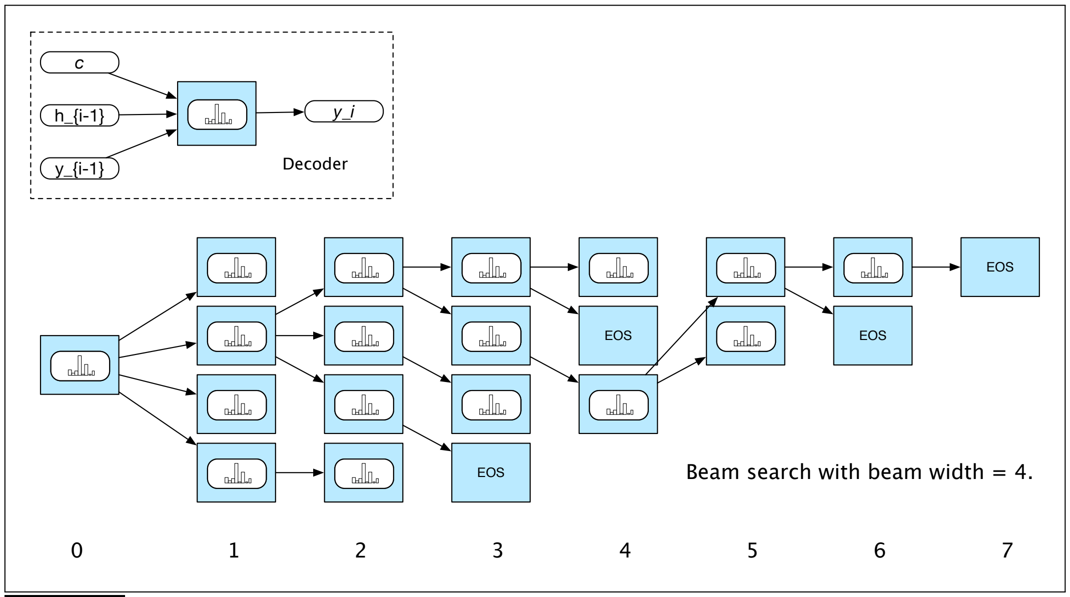

Figure 10.5 Beam decoding with a beam width of 4. At the initial step, the frontier is filled with the best 4 options from the initial state of the decoder. In a breadth-first fashion, each state on the frontier is passed to a decoder which computes a softmax over the entire vocabulary and attempts to enter each as a new state into the frontier subject to the constraint that they are better than the worst state already there. As completed sequences are discovered they are recorded and removed from the frontier and the beam width is reduced by 1.

One final complication arises from the fact that the completed hypotheses may have different lengths. Unfortunately, due to the probabilistic nature of our scoring scheme, longer hypotheses will naturally look worse than shorter ones just based on their length. This was not an issue during the earlier steps of decoding; due to the breadth-first nature of beam search all the hypotheses being compared had the same length. The usual solution to this is to apply some form of length normalization to each of the hypotheses. With normalization, we have B hypotheses and can select the best one, or we can pass all or a subset of them on to a downstream application with their respective scores.

Context

We've defined the context vector c as a function of the hidden states of the encoder, that is, $c = f(h_1^n)$. Unfortunately, the number of hidden states varies with the size of the input, making it difficult to just use them directly as a context for the decode. The basic approach described earlier avoids this issue since c is just the final hidden state

```
function BEAMDECODE($c$, beam_width) returns best paths

  $y_0$, $h_0 \leftarrow 0$
  path $\leftarrow$ ()
  complete_paths $\leftarrow$ ()
  state $\leftarrow$ ($c$, $y_0$, $h_0$, path)          ;initial state
  frontier $\leftarrow \langle$state$\rangle$        ;initial frontier

  while frontier contains incomplete paths and beamwidth $> 0$
      extended_frontier $\leftarrow \langle\rangle$
      for each state $\in$ frontier do
          $y \leftarrow$ DECODE(state)
          for each word $i \in$ Vocabulary do
              successor $\leftarrow$ NEWSTATE(state, $i$, $y_i$)
              new_agenda $\leftarrow$ ADDTOBEAM(successor, extended_frontier, beam_width)

      for each state in extended_frontier do
          if state is complete do
              complete_paths $\leftarrow$ APPEND(complete_paths, state)
              extended_frontier $\leftarrow$ REMOVE(extended_frontier, state)
              beam_width $\leftarrow$ beam_width - 1
      frontier $\leftarrow$ extended_frontier

    return completed_paths

function NEWSTATE(state, word, word_prob) returns new state

function ADDTOBEAM(state, frontier, width) returns updated frontier

  if LENGTH(frontier) $<$ width then
      frontier $\leftarrow$ INSERT(state, frontier)
  else if SCORE(state) $>$ SCORE(WORSTOF(frontier))
      frontier $\leftarrow$ REMOVE(WORSTOF(frontier))
      frontier $\leftarrow$ INSERT(state, frontier)
  return frontier
```

Figure 10.6 Beam search decoding.

of the encoder. This approach has the advantage of being simple and of reducing the context to a fixed length vector. However, this final hidden state inevitably is more focused on the latter parts of input sequence, rather than the input as whole.

One solution to this problem is to use Bi-RNNs, where the context can be a function of the end state of both the forward and backward passes. As described in Chapter 9, a straightforward approach is to concatenate the final states of the forward and backward passes. An alternative is to simply sum or average the encoder hidden states to produce a context vector. Unfortunately, this approach loses useful information about each of the individual encoder states that might prove useful in decoding.

10.3 Attention

To overcome the deficiencies of these simple approaches to context, we'll need a mechanism that can take the entire encoder context into account, that dynamically updates during the course of decoding, and that can be embodied in a fixed-size vector. Taken together, we'll refer such an approach as an **attention mechanism**.

attention mechanism

Our first step is to replace the static context vector with one that is dynamically derived from the encoder hidden states at each point during decoding. This context vector, c_i, is generated anew with each decoding step i and takes all of the encoder hidden states into account in its derivation. We then make this context available during decoding by conditioning the computation of the current decoder state on it, along with the prior hidden state and the previous output generated by the decoder.

$$h_i^d = g(\hat{y}_{i-1}, h_{i-1}^d, c_i)$$

The first step in computing c_i is to compute a vector of scores that capture the relevance of each encoder hidden state to the decoder state captured in h_{i-1}^d. That is, at each state i during decoding we'll compute $score(h_{i-1}^d, h_j^e)$ for each encoder state j.

For now, let's assume that this score provides us with a measure of how similar the decoder hidden state is to each encoder hidden state. To implement this similarity score, let's begin with the straightforward approach introduced in Chapter 6 of using the dot product between vectors.

$$score(h_{i-1}^d, h_j^e) = h_{i-1}^d \cdot h_j^e$$

The result of the dot product is a scalar that reflects the degree of similarity between the two vectors. And the vector of scores over all the encoder hidden states gives us the relevance of each encoder state to the current step of the decoder.

While the simple dot product can be effective, it is a static measure that does not facilitate adaptation during the course of training to fit the characteristics of given applications. A more robust similarity score can be obtained by parameterizing the score with its own set of weights, W_s.

$$score(h_{i-1}^d, h_j^e) = h_{t-1}^d W_s h_j^e$$

By introducing W_s to the score, we are giving the network the ability to learn which aspects of similarity between the decoder and encoder states are important to the current application.

To make use of these scores, we'll next normalize them with a softmax to create a vector of weights, α_{ij}, that tells us the proportional relevance of each encoder hidden state j to the current decoder state, i.

$$\alpha_{ij} = \text{softmax}(score(h_{i-1}^d, h_j^e) \ \forall j \in e)$$

$$= \frac{exp(score(h_{i-1}^d, h_j^e)}{\sum_k exp(score(h_{i-1}^d, h_k^e))}$$

Finally, given the distribution in α, we can compute a fixed-length context vector for the current decoder state by taking a weighted average over all the encoder hidden states.

$$c_i = \sum_j \alpha_{ij} h_j^e \qquad (10.1)$$

With this, we finally have a fixed-length context vector that takes into account information from the entire encoder state that is dynamically update to reflect the needs of the decoder at each step of decoding. Fig. 10.7 illustrates an encoder-decoder network with attention.

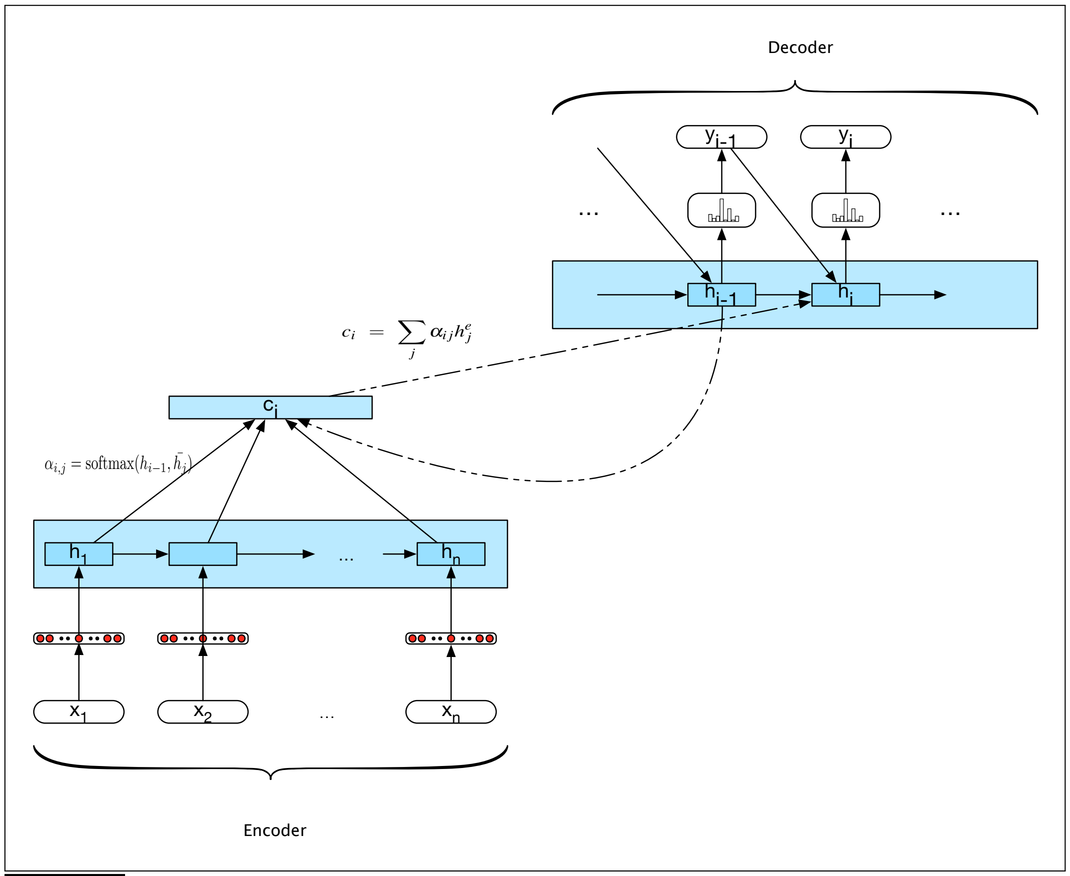

Figure 10.7 Encoder-decoder network with attention. Computing the value for h_i is based on the previous hidden state, the previous word generated, and the current context vector c_i. This context vector is derived from the attention computation based on comparing the previous hidden state to all of the encoder hidden states.

10.4 Applications of Encoder-Decoder Networks

The addition of attention to basic encoder-decoder networks led to rapid improvement in performance across a wide-range of applications including summarization, sentence simplification, question answering and image captioning.

10.5 Self-Attention and Transformer Networks

10.6 Summary

- Encoder-decoder networks
- Attention
- Transformers

Bibliographical and Historical Notes

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *EMNLP 2014*, 1724–1734.

Graves, A. (2013). Generating sequences with recurrent neural networks..

Graves, A., Fernández, S., Gomez, F., and Schmidhuber, J. (2006). Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, 369–376.

Graves, A., Fernández, S., Liwicki, M., Bunke, H., and Schmidhuber, J. (2007). Unconstrained on-line handwriting recognition with recurrent neural networks. In *NIPS 2007*, 577–584.

Kalchbrenner, N. and Blunsom, P. (2013). Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Luong, T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 1412–1421.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, 3104–3112.