CHAPTER 25

DEEP LEARNING FOR NATURAL LANGUAGE PROCESSING

In which deep neural networks perform a variety of language tasks, capturing the structure of natural language as well as its fluidity.

Chapter 24 explained the key elements of natural language, including grammar and semantics. Systems based on parsing and semantic analysis have demonstrated success on many tasks, but their performance is limited by the endless complexity of linguistic phenomena in real text. Given the vast amount of text available in machine-readable form, it makes sense to consider whether approaches based on data-driven machine learning can be more effective. We explore this hypothesis using the tools provided by deep learning systems (Chapter 22).

We begin in Section 25.1 by showing how learning can be improved by representing words as points in a high-dimensional space, rather than as atomic values. Section 25.2 covers the use of recurrent neural networks to capture meaning and long-distance context as text is processed sequentially. Section 25.3 focuses primarily on machine translation, one of the major successes of deep learning applied to NLP. Sections 25.4 and 25.5 cover models that can be trained from large amounts of unlabeled text and then applied to specific tasks, often achieving state-of-the-art performance. Finally, Section 25.6 takes stock of where we are and how the field may progress.

25.1 Word Embeddings

We would like a representation of words that does not require manual feature engineering, but allows for generalization between related words—words that are related syntactically ("colorless" and "ideal" are both adjectives), semantically ("cat" and "kitten" are both felines), topically ("sunny" and "sleet" are both weather terms), in terms of sentiment ("awesome" has opposite sentiment to "cringeworthy"), or otherwise.

How should we encode a word into an input vector $\mathbf{x}$ for use in a neural network? As explained in Section 22.2.1 (page 807), we could use a **one-hot vector**—that is, we encode the ith word in the dictionary with a 1 bit in the ith input position and a 0 in all the other positions. But such a representation would not capture the similarity between words.

Following the linguist John R. Firth's (1957) maxim, "You shall know a word by the company it keeps," we could represent each word with a vector of n-gram counts of all the phrases that the word appears in. However, raw n-gram counts are cumbersome. With a 100,000-word vocabulary, there are 10^{25} 5-grams to keep track of (although vectors in this 10^{25}-dimensional space would be quite sparse—most of the counts would be zero). We would get better gen-

Word embedding

eralization if we reduced this to a smaller-size vector, perhaps with just a few hundred dimensions. We call this smaller, dense vector a **word embedding**: a low-dimensional vector representing a word. Word embeddings are *learned automatically* from the data. (We will see later how this is done.) What are these learned word embeddings like? On the one hand, each one is just a vector of numbers, where the individual dimensions and their numeric values do not have discernible meanings:

$$\begin{aligned}
&\text{“aardvark”} = [-0.7, +0.2, -3.2, \ldots] \\
&\text{“abacus”} = [+0.5, +0.9, -1.3, \ldots] \\
&\ldots \\
&\text{“zyzzyva”} = [-0.1, +0.8, -0.4, \ldots].
\end{aligned}$$

On the other hand, the feature space has the property that similar words end up having similar vectors. We can see that in Figure 25.1, where there are separate clusters for country, kinship, transportation, and food words.

It turns out, for reasons we do not completely understand, that the word embedding vectors have additional properties beyond mere proximity for similar words. For example, suppose we look at the vectors $\mathbf{A}$ for Athens and $\mathbf{B}$ for Greece. For these words the vector difference $\mathbf{B}-\mathbf{A}$ seems to encode the country/capital relationship. Other pairs—France and Paris, Russia and Moscow, Zambia and Lusaka—have essentially the same vector difference.

We can use this property to solve word analogy problems such as “Athens is to Greece as Oslo is to [what]?” Writing $\mathbf{C}$ for the Oslo vector and $\mathbf{D}$ for the unknown, we hypothesize that $\mathbf{B}-\mathbf{A}=\mathbf{D}-\mathbf{C}$, giving us $\mathbf{D}=\mathbf{C}+(\mathbf{B}-\mathbf{A})$. And when we compute this new vector $\mathbf{D}$, we find that it is closer to “Norway” than to any other word. Figure 25.2 shows that this type of vector arithmetic works for many relationships.

However, there is no guarantee that a particular word embedding algorithm run on a particular corpus will capture a particular semantic relationship. Word embeddings are popular because they have proven to be a good representation for downstream language tasks (such as question answering or translation or summarization), not because they are guaranteed to answer analogy questions on their own.

Using word embedding vectors rather than one-hot encodings of words turns out to be helpful for essentially all applications of deep learning to NLP tasks. Indeed, in many cases it is possible to use generic **pretrained** vectors, obtained from any of several suppliers, for one’s particular NLP task. At the time of writing, the commonly used vector dictionaries include WORD2VEC, GloVe (Global Vectors), and FASTTEXT, which has embeddings for 157 languages. Using a pretrained model can save a great deal of time and effort. For more on these resources, see Section 25.5.1.

It is also possible to train your own word vectors; this is usually done at the same time as training a network for a particular task. Unlike generic pretrained embeddings, word embeddings produced for a specific task can be trained on a carefully selected corpus and will tend to emphasize aspects of words that are useful for the task. Suppose, for example, that the task is part-of-speech (POS) tagging (see Section 24.1.6). Recall that this involves predicting the correct part of speech for each word in a sentence. Although this is a simple task, it is nontrivial because many words can be tagged in multiple ways—for example, the word *cut* can be a present-tense verb (transitive or intransitive), a past-tense verb, an infinitive verb, a past participle, an adjective, or a noun. If a nearby temporal adverb refers to the past, that

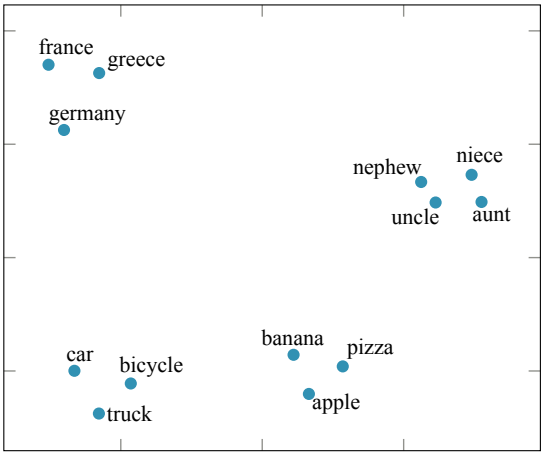

Figure 25.1 Word embedding vectors computed by the GloVe algorithm trained on 6 billion words of text. 100-dimensional word vectors are projected down onto two dimensions in this visualization. Similar words appear near each other.

A	**B**	**C**	**D = C + (B − A)**	**Relationship**
Athens	Greece	Oslo	Norway	*Capital*
Astana	Kazakhstan	Harare	Zimbabwe	*Capital*
Angola	kwanza	Iran	rial	*Currency*
copper	Cu	gold	Au	*Atomic Symbol*
Microsoft	Windows	Google	Android	*Operating System*
New York	New York Times	Baltimore	Baltimore Sun	*Newspaper*
Berlusconi	Silvio	Obama	Barack	*First name*
Switzerland	Swiss	Cambodia	Cambodian	*Nationality*
Einstein	scientist	Picasso	painter	*Occupation*
brother	sister	grandson	granddaughter	*Family Relation*
Chicago	Illinois	Stockton	California	*State*
possibly	impossibly	ethical	unethical	*Negative*
mouse	mice	dollar	dollars	*Plural*
easy	easiest	lucky	luckiest	*Superlative*
walking	walked	swimming	swam	*Past tense*

Figure 25.2 A word embedding model can sometimes answer the question "**A** is to **B** as **C** is to [what]?" with vector arithmetic: given the word embedding vectors for the words **A**, **B**, and **C**, compute the vector $\mathbf{D} = \mathbf{C} + (\mathbf{B} - \mathbf{A})$ and look up the word that is closest to **D**. (The answers in column **D** were computed automatically by the model. The descriptions in the "Relationship" column were added by hand.) Adapted from Mikolov *et al.* (2013, 2014).

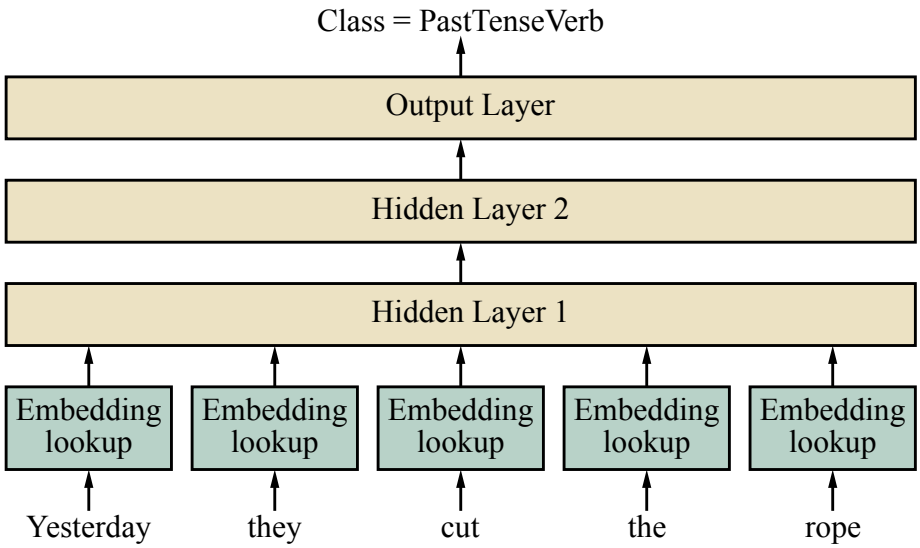

Figure 25.3 Feedforward part-of-speech tagging model. This model takes a 5-word window as input and predicts the tag of the word in the middle—here, *cut*. The model is able to account for word position because each of the 5 input embeddings is multiplied by a different part of the first hidden layer. The parameter values for the word embeddings and for the three layers are all learned simultaneously during training.

suggests that this particular occurrence of *cut* is a past-tense verb; and we might hope, then, that the embedding will capture the past-referring aspect of adverbs.

POS tagging serves as a good introduction to the application of deep learning to NLP, without the complications of more complex tasks like question answering (see Section 25.5.3). Given a corpus of sentences with POS tags, we learn the parameters for the word embeddings and the POS tagger simultaneously. The process works as follows:

1. Choose the width w (an odd number of words) for the prediction window to be used to tag each word. A typical value is $w = 5$, meaning that the tag is predicted based on the word plus the two words to the left and the two words to the right. Split every sentence in your corpus into overlapping windows of length w. Each window produces one training example consisting of the w words as input and the POS category of the middle word as output.
2. Create a vocabulary of all of the unique word tokens that occur more than, say, 5 times in the training data. Denote the total number of words in the vocabulary as v.
3. Sort this vocabulary in any arbitrary order (perhaps alphabetically).
4. Choose a value d as the size of each word embedding vector.
5. Create a new v-by-d weight matrix called $\mathbf{E}$. This is the word embedding matrix. Row i of $\mathbf{E}$ is the word embedding of the ith word in the vocabulary. Initialize $\mathbf{E}$ randomly (or from pretrained vectors).
6. Set up a neural network that outputs a part of speech label, as shown in Figure 25.3. The first layer will consist of w copies of the embedding matrix. We might use two additional hidden layers, $\mathbf{z}_1$ and $\mathbf{z}_2$ (with weight matrices $\mathbf{W}_1$ and $\mathbf{W}_2$, respectively), followed by

a softmax layer yielding an output probability distribution $\hat{\mathbf{y}}$ over the possible part-of-speech categories for the middle word:

$$\begin{aligned}\mathbf{z}_1 &= \sigma(\mathbf{W}_1\mathbf{x})\\ \mathbf{z}_2 &= \sigma(\mathbf{W}_2\mathbf{z}_1)\\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{W}_{out}\mathbf{z}_2)\,.\end{aligned}$$

7. To encode a sequence of w words into an input vector, simply look up the embedding for each word and concatenate the embedding vectors. The result is a real-valued input vector $\mathbf{x}$ of length wd. Even though a given word will have the same embedding vector whether it occurs in the first position, the last, or somewhere in between, each embedding will be multiplied by a different part of the first hidden layer; therefore we are implicitly encoding the relative position of each word.
8. Train the weights $\mathbf{E}$ and the other weight matrices $\mathbf{W}_1$, $\mathbf{W}_2$, and $\mathbf{W}_{out}$ using gradient descent. If all goes well, the middle word, *cut*, will be labeled as a past-tense verb, based on the evidence in the window, which includes the temporal past word "yesterday," the third-person subject pronoun "they" immediately before *cut*, and so on.

An alternative to word embeddings is a **character-level model** in which the input is a sequence of characters, each encoded as a one-hot vector. Such a model has to learn how characters come together to form words. The majority of work in NLP sticks with word-level rather than character-level encodings.

25.2 Recurrent Neural Networks for NLP

We now have a good representation for single words in isolation, but language consists of an ordered sequence of words in which the **context** of surrounding words is important. For simple tasks like part of speech tagging, a small, fixed-size window of perhaps five words usually provides enough context.

More complex tasks such as question answering or reference resolution may require dozens of words as context. For example, in the sentence "*Eduardo told me that Miguel was very sick so I took* **him** *to the hospital*," knowing that **him** refers to *Miguel* and not *Eduardo* requires context that spans from the first to the last word of the 14-word sentence.

25.2.1 Language models with recurrent neural networks

We'll start with the problem of creating a **language model** with sufficient context. Recall that a language model is a probability distribution over sequences of words. It allows us to predict the next word in a text given all the previous words, and is often used as a building block for more complex tasks.

Building a language model with either an n-gram model (as in Section 24.1) or a feedforward network with a fixed window of n words can run into difficulty due to the problem of context: either the required context will exceed the fixed window size or the model will have too many parameters, or both.

In addition, a feedforward network has the problem of **asymmetry**: whatever it learns about, say, the appearance of the word *him* as the 12th word of the sentence it will have to relearn for the appearance of *him* at other positions in the sentence, because the weights are different for each word position.

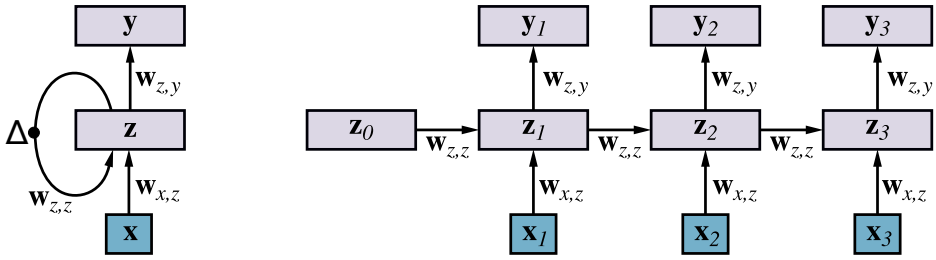

Figure 25.4 (a) Schematic diagram of an RNN where the hidden layer $\mathbf{z}$ has recurrent connections; the Δ symbol indicates a delay. Each input $\mathbf{x}$ is the word embedding vector of the next word in the sentence. Each output $\mathbf{y}$ is the output for that time step. (b) The same network unrolled over three timesteps to create a feedforward network. Note that the weights are shared across all timesteps.

In Section 22.6, we introduced the **recurrent neural network** or **RNN**, which is designed to process time-series data, one datum at a time. This suggests that RNNs might be useful for processing language, one word at a time. We repeat Figure 22.8 here as Figure 25.4.

In an RNN language model each input word is encoded as a word embedding vector, $\mathbf{x}_i$. There is a hidden layer $\mathbf{z}_t$ which gets passed as input from one time step to the next. We are interested in doing multiclass classification: the classes are the words of the vocabulary. Thus the output $\mathbf{y}_t$ will be a softmax probability distribution over the possible values of the next word in the sentence.

The RNN architecture solves the problem of too many parameters. The number of parameters in the weight matrixes $w_{,z,z}$, $w_{,x,z}$, and $w_{,z,y}$ stays constant, regardless of the number of words—it is $O(1)$. This is in contrast to feedforward networks, which have $O(n)$ parameters, and n-gram models, which have $O(v^n)$ parameters, where v is the size of the vocabulary.

The RNN architecture also solves the problem of asymmetry, because the weights are the same for every word position.

The RNN architecture can sometimes solve the limited context problem as well. In theory there is no limit to how far back in the input the model can look. Each update of the hidden layer $\mathbf{z}_t$ has access to both the current input word $\mathbf{x}_t$ and the previous hidden layer $\mathbf{z}_{t-1}$, which means that information about any word in the input can be kept in the hidden layer indefinitely, copied over (or modified as appropriate) from one time step to the next. Of course, there is a limited amount of storage in $\mathbf{z}$, so it can't remember everything about all the previous words.

In practice RNN models perform well on a variety of tasks, but not on all tasks. It can be hard to predict whether they will be succesful for a given problem. One factor that contributes to success is that the training process encourages the network to allocate storage space in $\mathbf{z}$ to the aspects of the input that will actually prove to be useful.

To train an RNN language model, we use the training process described in Section 22.6.1. The inputs, $\mathbf{x}_t$, are the words in a training corpus of text, and the observed outputs are the same

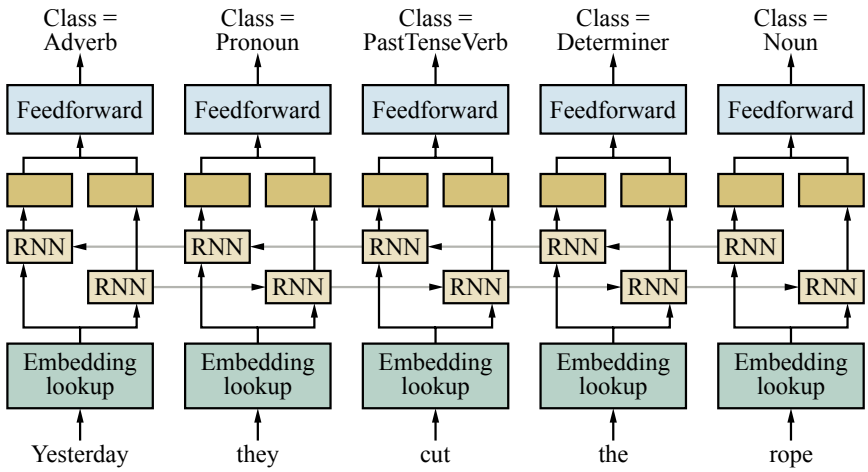

Figure 25.5 A bidirectional RNN network for POS tagging.

words offset by 1. That is, for the training text "hello world," the first input $\mathbf{x}_1$ is the word embedding for "hello" and the first output $\mathbf{y}_1$ is the word embedding for "world." We are training the model to predict the next word, and expecting that in order to do so it will use the hidden layer to represent useful information. As explained in Section 22.6.1 we compute the difference between the observed output and the actual output computed by the network, and back-propagate through time, taking care to keep the weights the same for all time steps.

Once the model has been trained, we can use it to generate random text. We give the model an initial input word $\mathbf{x}_1$, from which it will produce an output $\mathbf{y}_1$ which is a softmax probability distribution over words. We sample a single word from the distribution, record the word as the output for time t, and feed it back in as the next input word $\mathbf{x}_2$. We repeat for as long as desired. In sampling from $\mathbf{y}_1$ we have a choice: we could always take the most likely word; we could sample according to the probability of each word; or we could oversample the less-likely words, in order to inject more variety into the generated output. The sampling weight is a hyperparameter of the model.

Here is an example of random text generated by an RNN model trained on Shakespeare's works (Karpathy, 2015):

> *Marry, and will, my lord, to weep in such a one were prettiest;*
> *Yet now I was adopted heir*
> *Of the world's lamentable day,*
> *To watch the next way with his father with his face?*

25.2.2 Classification with recurrent neural networks

It is also possible to use RNNs for other language tasks, such as part of speech tagging or coreference resolution. In both cases the input and hidden layers will be the same, but for a POS tagger the output will be a softmax distribution over POS tags, and for coreference resolution it will be a softmax distribution over the possible antecedents. For example, when the network gets to the input **him** in "*Eduardo told me that Miguel was very sick so I took* **him** *to the hospital*" it should output a high probability for "*Miguel*."

Training an RNN to do classification like this is done the same way as with the language model. The only difference is that the training data will require labels—part of speech tags or reference indications. That makes it much harder to collect the data than for the case of a language model, where unlabelled text is all we need.

In a language model we want to predict the nth word given the previous words. But for classification, there is no reason we should limit ourselves to looking at only the previous words. It can be very helpful to look ahead in the sentence. In our coreference example, the referent *him* would be different if the sentence concluded "to see Miguel" rather than "to the hospital," so looking ahead is crucial. We know from eye-tracking experiments that human readers do not go strictly left-to-right.

Bidirectional RNN

To capture the context on the right, we can use a **bidirectional RNN**, which concatenates a separate right-to-left model onto the left-to-right model. An example of using a bidirectional RNN for POS tagging is shown in Figure 25.5.

In the case of a multilayer RNN, $\mathbf{z}_t$ will be the hidden vector of the last layer. For a bidirectional RNN, $\mathbf{z}_t$ is usually taken to be the concatenation of vectors from the left-to-right and right-to-left models.

RNNs can also be used for sentence-level (or document-level) classification tasks, in which a single output comes at the end, rather than having a stream of outputs, one per time step. For example in **sentiment analysis** the goal is to classify a text as having either *Positive* or *Negative* sentiment. For example, *"This movie was poorly written and poorly acted"* should be classified as *Negative*. (Some sentiment analysis schemes use more than two categories, or use a numeric scalar value.)

Using RNNs for a sentence-level task is a bit more complex, since we need to obtain an aggregate whole-sentence representation, $\mathbf{y}$ from the per-word outputs $\mathbf{y}_t$ of the RNN. The simplest way to do this is to use the RNN hidden state corresponding to the last word of the input, since the RNN will have read the entire sentence at that timestep. However, this can implicitly bias the model towards paying more attention to the end of the sentence.

Average pooling

Another common technique is to pool all of the hidden vectors. For instance, **average pooling** computes the element-wise average over all of the hidden vectors:

$$\tilde{\mathbf{z}} = \frac{1}{s}\sum_{t=1}^{s}\mathbf{z}_t\,.$$

The pooled d-dimensional vector $\tilde{\mathbf{z}}$ can then be fed into one or more feedforward layers before being fed into the output layer.

25.2.3 LSTMs for NLP tasks

We said that RNNs sometimes solve the limited context problem. In theory, any information could be passed along from one hidden layer to the next for any number of time steps. But in practice the information can get lost or distorted, just as in playing the game of telephone, in which players stand in line and the first player whispers a message to the second, who repeats it to the third, and so on down the line. Usually, the message that comes out at the end is quite corrupted from the original message. This problem for RNNs is similar to the **vanishing gradient** problem we described on page 807, except that we are dealing now with layers over time rather than with deep layers.

In Section 22.6.2 we introduced the **long short-term memory (LSTM)** model. This is a kind of RNN with gating units that don't suffer from the problem of imperfectly reproducing

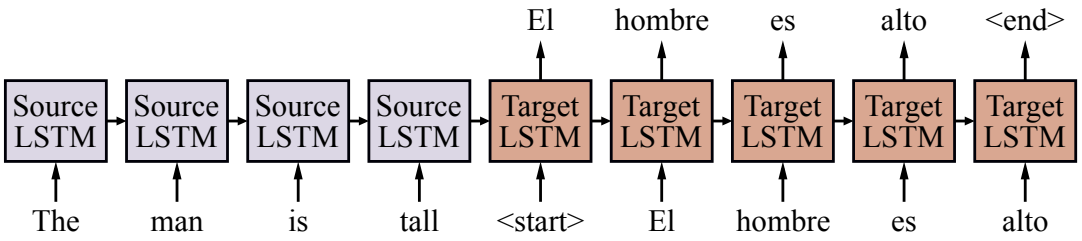

Figure 25.6 Basic sequence-to-sequence model. Each block represents one LSTM timestep. (For simplicity, the embedding and output layers are not shown.) On successive steps we feed the network the words of the source sentence "The man is tall," followed by the `<start>` tag to indicate that the network should start producing the target sentence. The final hidden state at the end of the source sentence is used as the hidden state for the start of the target sentence. After that, each target sentence word at time t is used as input at time $t+1$, until the network produces the `<end>` tag to indicate that sentence generation is finished.

a message from one time step to the next. Rather, an LSTM can choose to *remember* some parts of the input, copying it over to the next timestep, and to forget other parts. Consider a language model handling a text such as

The athletes, who all won their local qualifiers and advanced to the finals in Tokyo, now ...

At this point if we asked the model which next word was more probable, "compete" or "competes," we would expect it to pick "compete" because it agrees with the subject "The athletes." An LSTM can learn to create a latent feature for the subject person and number and copy that feature forward without alteration until it is needed to make a choice like this. A regular RNN (or an n-gram model for that matter) often gets confused in long sentences with many intervening words between the subject and verb.

25.3 Sequence-to-Sequence Models

Machine translation (MT)
Source language
Target language

One of the most widely studied tasks in NLP is **machine translation (MT)**, where the goal is to translate a sentence from a **source language** to a **target language**—for example, from Spanish to English. We train an MT model with a large corpus of source/target sentence pairs. The goal is to then accurately translate new sentences that are not in our training data.

Can we use RNNs to create an MT system? We can certainly encode the source sentence with an RNN. If there were a one-to-one correspondence between source words and target words, then we could treat MT as a simple tagging task—given the source word "perro" in Spanish, we tag it as the corresponding English word "dog." But in fact, words are not one-to-one: in Spanish the three words "caballo de mar" corresponds to the single English word "seahorse," and the two words "perro grande" translate to "big dog," with the word order reversed. Word reordering can be even more extreme; in English the subject is usually at the start of a sentence, but in Fijian the subject is usually at the end. So how do we generate a sentence in the target language?

It seems like we should generate one word at a time, but keep track of the context so that we can remember parts of the source that haven't been translated yet, and keep track of what has been translated so that we don't repeat ourselves. It also seems that for some sentences

we have to process the entire source sentence before starting to generate the target. In other words, the generation of each target word is conditional on the entire source sentence and on all previously generated target words.

This gives text generation for MT a close connection to a standard RNN language model, as described in Section 25.2. Certainly, if we had trained an RNN on English text, it would be more likely to generate "big dog" than "dog big." However, we don't want to generate just any random target language sentence; we want to generate a target language sentence that *corresponds* to the source language sentence. The simplest way to do that is to use two RNNs, one for the source and one for the target. We run the source RNN over the source sentence and then use the final hidden state from the source RNN as the initial hidden state for the target RNN. This way, each target word is implicitly conditioned on both the entire source sentence and the previous target words.

Sequence-to-sequence model

This neural network architecture is called a basic **sequence-to-sequence model**, an example of which is shown in Figure 25.6. Sequence-to-sequence models are most commonly used for machine translation, but can also be used for a number of other tasks, like automatically generating a text caption from an image, or summarization: rewriting a long text into a shorter one that maintains the same meaning.

Basic sequence-to-sequence models were a significant breakthrough in NLP and MT specifically. According to Wu *et al.* (2016b) the approach led to a 60% error reduction over the previous MT methods. But these models suffer from three major shortcomings:

- **Nearby context bias**: whatever RNNs want to remember about the past, they have to fit into their hidden state. For example, let's say an RNN is processing word (or timestep) 57 in a 70-word sequence. The hidden state will likely contain more information about the word at timestep 56 than the word at timestep 5, because each time the hidden vector is updated it has to replace some amount of existing information with new information. This behavior is part of the intentional design of the model, and often makes sense for NLP, since nearby context is typically more important. However, far-away context can be crucial as well, and can get lost in an RNN model; even LSTMs have difficulty with this task.
- **Fixed context size limit**: In an RNN translation model the entire source sentence is compressed into a single fixed-dimensional hidden state vector. An LSTM used in a state-of-the-art NLP model typically has around 1024 dimensions, and if we have to represent, say, a 64-word sentence in 1024 dimensions, this only gives us 16 dimensions per word—not enough for complex sentences. Increasing the hidden state vector size can lead to slow training and overfitting.
- **Slower sequential processing**: As discussed in Section 22.3, neural networks realize considerable efficiency gains by processing the training data in batches so as to take advantage of efficient hardware support for matrix arithmetic. RNNs, on the other hand, seem to be constrained to operate on the training data one word at a time.

25.3.1 Attention

What if the target RNN were conditioned on *all* of the hidden vectors from the source RNN, rather than just the last one? This would mitigate the shortcomings of nearby context bias and fixed context size limits, allowing the model to access any previous word equally well. One way to achieve this access is to concatenate all of the source RNN hidden vectors. However,

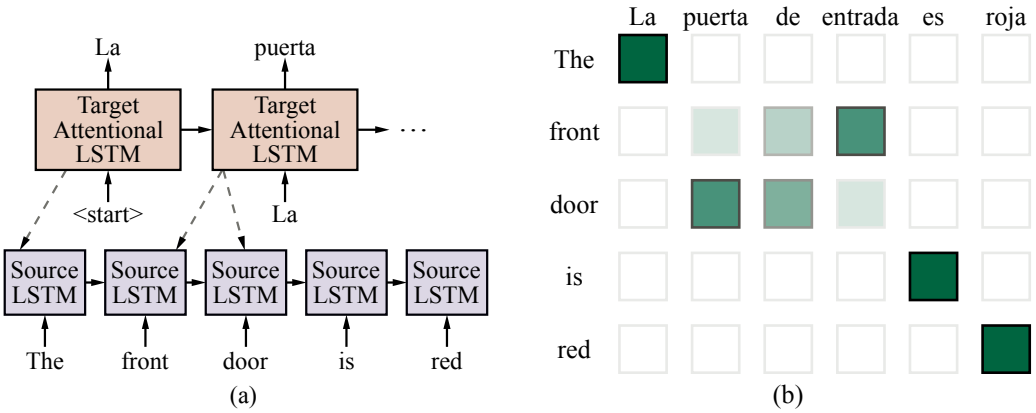

Figure 25.7 (a) Attentional sequence-to-sequence model for English-to-Spanish translation. The dashed lines represent attention. (b) Example of attention probability matrix for a bilingual sentence pair, with darker boxes representing higher values of a_{ij}. The attention probabilities sum to one over each column.

this would cause a huge increase in the number of weights, with a concomitant increase in computation time and potentially overfitting as well. Instead, we can take advantage of the fact that when the target RNN is generating the target one word at a time, it is likely that only a small part of the source is actually relevant to each target word.

Crucially, the target RNN must pay attention to different parts of the source for every word. Suppose a network is trained to translate English to Spanish. It is given the words "The front door is red" followed by an end of sentence marker, which means it is time to start outputting Spanish words. So ideally it should first pay attention to "The" and generate "La," then pay attention to "door" and output "puerta," and so on.

Attention

We can formalize this concept with a neural network component called **attention**, which can be used to create a "context-based summarization" of the source sentence into a fixed-dimensional representation. The context vector $\mathbf{c}_i$ contains the most relevant information for generating the next target word, and will be used as an additional input to the target RNN. A sequence-to-sequence model that uses attention is called an **attentional sequence-to-sequence model**. If the standard target RNN is written as:

Attentional sequence-to-sequence model

$$\mathbf{h}_i = RNN(\mathbf{h}_{i-1}, \mathbf{x}_i),$$

the target RNN for attentional sequence-to-sequence models can be written as:

$$\mathbf{h}_i = RNN(\mathbf{h}_{i-1}, [\mathbf{x}_i; \mathbf{c}_i])$$

where $[\mathbf{x}_i; \mathbf{c}_i]$ is the concatenation of the input and context vectors, $\mathbf{c}_i$, defined as:

$$\begin{aligned} r_{ij} &= \mathbf{h}_{i-1} \cdot \mathbf{s}_j \\ a_{ij} &= e^{r_{ij}} / (\sum_k e^{r_{ik}}) \\ \mathbf{c}_i &= \sum_j a_{ij} \cdot \mathbf{s}_j \end{aligned}$$

where $\mathbf{h}_{i-1}$ is the target RNN vector that is going to be used for predicting the word at timestep i, and $\mathbf{s}_j$ is the output of the source RNN vector for the source word (or timestep) j. Both $\mathbf{h}_{i-1}$ and $\mathbf{s}_j$ are d-dimensional vectors, where d is the hidden size. The value of r_{ij} is therefore the raw "attention score" between the current target state and the source word j. These scores are then normalized into a probability a_{ij} using a softmax over all source words. Finally, these probabilities are used to generate a weighted average of the source RNN vectors, $\mathbf{c}_i$ (another d-dimensional vector).

An example of an attentional sequence-to-sequence model is given in Figure 25.7 (a). There are a few important details to understand. First, the attention component itself has no learned weights and supports variable-length sequences on both the source and target side. Second, like most of the other neural network modeling techniques we've learned about, attention is entirely latent. The programmer does not dictate what information gets used when; the model learns what to use. Attention can also be combined with multilayer RNNs. Typically attention is applied at each layer in that case.

The probabilistic softmax formulation for attention serves three purposes. First, it makes attention differentiable, which is necessary for it to be used with back-propagation. Even though attention itself has no learned weights, the gradients still flow back through attention to the source and target RNNs. Second, the probabilistic formulation allows the model to capture certain types of long-distance contextualization that may have not been captured by the source RNN, since attention can consider the entire source sequence at once, and learn to keep what is important and ignore the rest. Third, probabilistic attention allows the network to represent uncertainty—if the network does not know exactly what source word to translate next, it can distribute the attention probabilities over several options, and then actually choose the word using the target RNN.

Unlike most components of neural networks, attention probabilities are often interpretable by humans and intuitively meaningful. For example, in the case of machine translation, the attention probabilities often correspond to the word-to-word alignments that a human would generate. This is shown in Figure 25.7(b).

Sequence-to-sequence models are a natural for machine translation, but almost any natural language task can be encoded as a sequence-to-sequence problem. For example, a question-answering system can be trained on input consisting of a question followed by a delimiter followed by the answer.

25.3.2 Decoding

At training time, a sequence-to-sequence model attempts to maximize the probability of each word in the target training sentence, conditioned on the source and all of the previous target words. Once training is complete, we are given a source sentence, and our goal is to generate the corresponding target sentence. As shown in Figure 25.7, we can generate the target one word at a time, and then feed back in the word that we generated at the next timestep. This procedure is called **decoding**.

Decoding

The simplest form of decoding is to select the highest probability word at each timestep and then feed this word as input to the next timestep. This is called **greedy decoding** because after each target word is generated, the system has fully committed to the hypothesis that it has produced so far. The problem is that the goal of decoding is to maximize the probability of the entire target sequence, which greedy decoding may not achieve. For example, consider

Greedy decoding

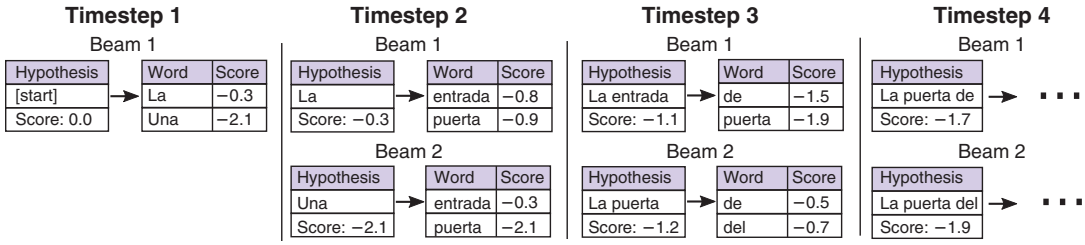

Figure 25.8 Beam search with beam size of $b = 2$. The score of each word is the log-probability generated by the target RNN softmax, and the score of each hypothesis is the sum of the word scores. At timestep 3, the highest scoring hypothesis *La entrada* can only generate low-probability continuations, so it "falls off the beam."

using a greedy decoder to translate into Spanish the English sentence we saw before, *The front door is red.*

The correct translation is *"La puerta de entrada es roja"*—literally *"The door of entry is red."* Suppose the target RNN correctly generates the first word *La* for *The*. Next, a greedy decoder might propose *entrada* for *front*. But this is an error—Spanish word order should put the noun *puerta* before the modifier. Greedy decoding is fast—it only considers one choice at each timestep and can do so quickly—but the model has no mechanism to correct mistakes.

We could try to improve the attention mechanism so that it always attends to the right word and guesses correctly every time. But for many sentences it is infeasible to guess correctly all the words at the start of the sentence until you have seen what's at the end.

A better approach is to search for an optimal decoding (or at least a good one) using one of the search algorithms from Chapter 3. A common choice is a **beam search** (see Section 4.1.3). In the context of MT decoding, beam search typically keeps the top k hypotheses at each stage, extending each by one word using the top k choices of words, then chooses the best k of the resulting k^2 new hypotheses. When all hypotheses in the beam generate the special `<end>` token, the algorithm outputs the highest scoring hypothesis.

A visualization of beam search is given in Figure 25.8. As deep learning models become more accurate, we can usually afford to use a smaller beam size. Current state-of-the-art neural MT models use a beam size of 4 to 8, whereas the older generation of statistical MT models would use a beam size of 100 or more.

25.4 The Transformer Architecture

The influential article "Attention is all you need" (Vaswani *et al.*, 2018) introduced the **transformer** architecture, which uses a **self-attention** mechanism that can model long-distance context without a sequential dependency.

Transformer
Self-attention

25.4.1 Self-attention

Previously, in sequence-to-sequence models, attention was applied from the target RNN to the source RNN. **Self-attention** extends this mechanism so that each sequence of hidden states also attends to itself—the source to the source, and the target to the target. This allows the model to additionally capture long-distance (and nearby) context within each sequence.

Self-attention

The most straightforward way of applying self-attention is where the attention matrix is directly formed by the dot product of the input vectors. However, this is problematic. The dot product between a vector and itself will always be high, so each hidden state will be biased towards attending to itself. The transformer solves this by first projecting the input into three different representations using three different weight matrices:

Query vector
- The **query vector** $\mathbf{q}_i = \mathbf{W}_q \mathbf{x}_i$ is the one being *attended from*, like the target in the standard attention mechanism.

Key vector
- The **key vector** $\mathbf{k}_i = \mathbf{W}_k \mathbf{x}_i$ is the one being *attended to*, like the source in the basic attention mechanism.

Value vector
- The **value vector** $\mathbf{v}_i = \mathbf{W}_v \mathbf{x}_i$ is the context that is being generated.

In the standard attention mechanism, the key and value networks are identical, but intuitively it makes sense for these to be separate representations. The encoding results of the ith word, $\mathbf{c}_i$, can be calculated by applying an attention mechanism to the projected vectors:

$$\begin{aligned} r_{ij} &= (\mathbf{q}_i \cdot \mathbf{k}_j)/\sqrt{d} \\ a_{ij} &= e^{r_{ij}}/(\sum_k e^{r_{ik}}) \\ \mathbf{c}_i &= \sum_j a_{ij} \cdot \mathbf{v}_j , \end{aligned}$$

where d is the dimension of $\mathbf{k}$ and $\mathbf{q}$. Note that i and j are indexes in the same sentence, since we are encoding the context using self-attention. In each transformer layer, self-attention uses the hidden vectors from the previous layer, which initially is the embedding layer.

There are several details worth mentioning here. First of all, the self-attention mechanism is *asymmetric*, as r_{ij} is different from r_{ji}. Second, the scale factor $\sqrt{d}$ was added to improve numerical stability. Third, the encoding for all words in a sentence can be calculated simultaneously, as the above equations can be expressed using matrix operations that can be computed efficiently in parallel on modern specialized hardware.

The choice of which context to use is completely learned from training examples, not prespecified. The context-based summarization, $\mathbf{c}_i$, is a sum over all previous positions in the sentence. In theory, any information from the sentence could appear in $\mathbf{c}_i$, but in practice, sometimes important information gets lost, because it is essentially averaged out over the whole sentence. One way to address that is called **multiheaded attention**. We divide the sentence up into m equal pieces and apply the attention model to each of the m pieces. Each piece has its own set of weights. Then the results are concatenated together to form $\mathbf{c}_i$. By concatenating rather than summing, we make it easier for an important subpiece to stand out.

Multiheaded attention

25.4.2 From self-attention to transformer

Self-attention is only one component of the transformer model. Each transformer layer consists of several sub-layers. At each transformer layer, self-attention is applied first. The output of the attention module is fed through feedforward layers, where the same feedforward weight matrices are applied independently at each position. A nonlinear activation function, typically ReLU, is applied after the first feedforward layer. In order to address the potential vanishing gradient problem, two residual connections are added into the transformer layer. A single-layer transformer in shown in Figure 25.9. In practice, transformer models usually

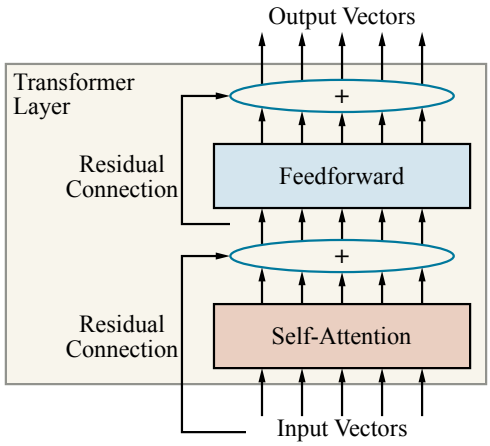

Figure 25.9 A single-layer transformer consists of self-attention, a feedforward network, and residual connections.

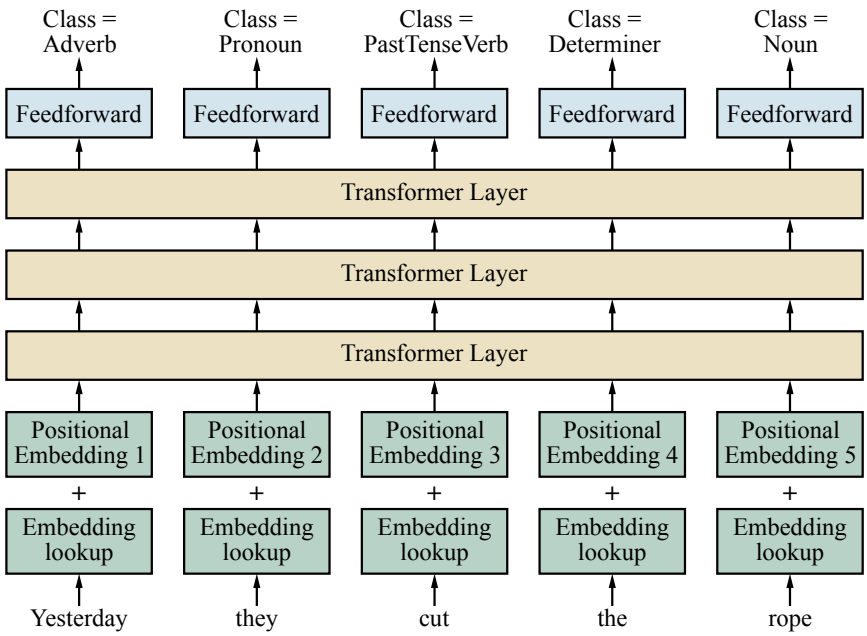

Figure 25.10 Using the transformer architecture for POS tagging.

have six or more layers. As with the other models that we've learned about, the output of layer i is used as the input to layer $i+1$.

The transformer architecture does not explicitly capture the order of words in the sequence, since context is modeled only through self-attention, which is agnostic to word order. To capture the ordering of the words, the transformer uses a technique called **positional embedding**. If our input sequence has a maximum length of n, then we learn n new embedding

Positional embedding

vectors—one for each word position. The input to the first transformer layer is the sum of the word embedding at position t plus the positional embedding corresponding to position t.

Figure 25.10 illustrates the transformer architecture for POS tagging, applied to the same sentence used in Figure 25.3. At the bottom, the word embedding and the positional embeddings are summed to form the input for a three-layer transformer. The transformer produces one vector per word, as in RNN-based POS tagging. Each vector is fed into a final output layer and softmax layer to produce a probability distribution over the tags.

Transformer encoder

In this section, we have actually only told half the transformer story: the model we described here is called the **transformer encoder**. It is useful for text classification tasks. The full transformer architecture was originally designed as a sequence-to-sequence model for machine translation. Therefore, in addition to the encoder, it also includes a **transformer decoder**. The encoder and decoder are nearly identical, except that the decoder uses a version of self-attention where each word can only attend to the words before it, since text is generated left-to-right. The decoder also has a second attention module in each transformer layer that attends to the output of the transformer encoder.

Transformer decoder

25.5 Pretraining and Transfer Learning

Getting enough data to build a robust model can be a challenge. In computer vision (see Chapter 27), that challenge was addressed by assembling large collections of images (such as ImageNet) and hand-labeling them.

For natural language, it is more common to work with text that is unlabeled. The difference is in part due to the difficulty of labeling: an unskilled worker can easily label an image as "cat" or "sunset," but it requires extensive training to annotate a sentence with part-of-speech tags or parse trees. The difference is also due to the abundance of text: the Internet adds over 100 billion words of text each day, including digitized books, curated resources such as Wikipedia, and uncurated social media posts.

Projects such as Common Crawl provide easy access to this data. Any running text can be used to build n-gram or word embedding models, and some text comes with structure that can be helpful for a variety of tasks—for example, there are many FAQ sites with question-answer pairs that can be used to train a question-answering system. Similarly, many Web sites publish side-by-side translations of texts, which can be used to train machine translation systems. Some text even comes with labels of a sort, such as review sites where users annotate their text reviews with a 5-star rating system.

Pretraining

We would prefer not to have to go to the trouble of creating a new data set every time we want a new NLP model. In this section, we introduce the idea of **pretraining**: a form of **transfer learning** (see Section 22.7.2) in which we use a large amount of shared general-domain language data to train an initial version of an NLP model. From there, we can use a smaller amount of domain-specific data (perhaps including some labeled data) to refine the model. The refined model can learn the vocabulary, idioms, syntactic structures, and other linguistic phenomena that are specific to the new domain.

25.5.1 Pretrained word embeddings

In Section 25.1, we briefly introduced word embeddings. We saw that how similar words like *banana* and *apple* end up with similar vectors, and we saw that we can solve analogy

problems with vector subtraction. This indicates that the word embeddings are capturing substantial information about the words.

In this section we will dive into the details of how word embeddings are created using an entirely unsupervised process over a large corpus of text. That is in contrast to the embeddings from Section 25.1, which were built during the process of supervised part of speech tagging, and thus required POS tags that come from expensive hand annotation.

We will concentrate on one specific model for word embeddings, the GloVe (Global Vectors) model. The model starts by gathering counts of how many times each word appears within a window of another word, similar to the skip-gram model. First choose window size (perhaps 5 words) and let X_{ij} be the number of times that words i and j co-occur within a window, and let X_i be the number of times word i co-occurs with any other word. Let $P_{ij} = X_{ij}/X_i$ be the probability that word j appears in the context of word i. As before, let $\mathbf{E}_i$ be the word embedding for word i.

Part of the intuition of the GloVe model is that the relationship between two words can best be captured by comparing them both to other words. Consider the words *ice* and *steam*. Now consider the ratio of their probabilities of co-occurrence with another word, w, that is:

$$P_{w,ice}/P_{w,steam}\,.$$

When w is the word *solid* the ratio will be high (meaning *solid* applies more to *ice*) and when w is the word *gas* it will be low (meaning *gas* applies more to *steam*). And when w is a non-content word like *the*, a word like *water* that is equally relevant to both, or an equally irrelevant word like *fashion*, the ratio will be close to 1.

The GloVe model starts with this intuition and goes through some mathematical reasoning (Pennington *et al.*, 2014) that converts ratios of probabilities into vector differences and dot products, eventually arriving at the constraint

$$\mathbf{E}_i \cdot \mathbf{E}'_k = \log(P_{ij})\,.$$

In other words, the dot product of two word vectors is equal to the log probability of their co-occurrence. That makes intuitive sense: two nearly-orthogonal vectors have a dot product close to 0, and two nearly-identical normalized vectors have a dot product close to 1. There is a technical complication wherein the GloVe model creates two word embedding vectors for each word, $\mathbf{E}_i$ and E'_i; computing the two and then adding them together at the end helps limit overfitting.

Training a model like GloVe is typically much less expensive than training a standard neural network: a new model can be trained from billions of words of text in a few hours using a standard desktop CPU.

It is possible to train word embeddings on a specific domain, and recover knowledge in that domain. For example, Tshitoyan *et al.* (2019) used 3.3 million scientific abstracts on the subject of material science to train a word embedding model. They found that, just as we saw that a generic word embedding model can answer "Athens is to Greece as Oslo is to what?" with "Norway," their material science model can answer "NiFe is to ferromagnetic as IrMn is to what?" with "antiferromagnetic."

Their model does not rely solely on co-occurrence of words; it seems to be capturing more complex scientific knowledge. When asked what chemical compounds can be classified as "thermoelectric" or "topological insulator," their model is able to answer correctly. For example, $CsAgGa_2Se_4$ never appears near "thermoelectric" in the corpus, but it does appear

near "chalcogenide," "band gap," and "optoelectric," which are all clues enabling it to be classified as similar to "thermoelectric." Furthermore, when trained only on abstracts up to the year 2008 and asked to pick compounds that are "thermoelectric" but have not yet appeared in abstracts, three of the model's top five picks were discovered to be thermoelectric in papers published between 2009 and 2019.

25.5.2 Pretrained contextual representations

Word embeddings are better representations than atomic word tokens, but there is an important issue with polysemous words. For example, the word *rose* can refer to a flower or the past tense of *rise*. Thus, we expect to find at least two entirely distinct clusters of word contexts for *rose*: one similar to flower names such as *dahlia*, and one similar to *upsurge*. No single embedding vector can capture both of these simultaneously. *Rose* is a clear example of a word with (at least) two distinct meanings, but other words have subtle shades of meaning that depend on context, such as the word *need* in *you need to see this movie* versus *humans need oxygen to survive*. And some idiomatic phrases like *break the bank* are better analyzed as a whole rather than as component words.

Contextual representations

Therefore, instead of just learning a word-to-embedding table, we want to train a model to generate **contextual representations** of each word in a sentence. A contextual representation maps both a word and the surrounding context of words into a word embedding vector. In other words, if we feed this model the word *rose* and the context *the gardener planted a rose bush*, it should produce a contextual embedding that is similar (but not necessarily identical) to the representation we get with the context *the cabbage rose had an unusual fragrance*, and very different from the representation of *rose* in the context *the river rose five feet*.

Figure 25.11 shows a recurrent network that creates contextual word embeddings—the boxes that are unlabeled in the figure. We assume we have already built a collection of noncontextual word embeddings. We feed in one word at a time, and ask the model to predict the next word. So for example in the figure at the point where we have reached the word "car," the the RNN node at that time step will receive two inputs: the noncontextual word embedding for "car" and the context, which encodes information from the previous words "The red." The RNN node will then output a contextual representation for "car." The network as a whole then outputs a prediction for the next word, "is." We then update the network's weights to minimize the error between the prediction and the actual next word.

This model is similar to the one for POS tagging in Figure 25.5, with two important differences. First, this model is unidirectional (left-to-right), whereas the POS model is bidirectional. Second, instead of predicting the POS tags for the *current* word, this model predicts the *next* word using the prior context. Once the model is built, we can use it to retrieve representations for words and pass them on to some other task; we need not continue to predict the next word. Note that computing a contextual representations always requires two inputs, the current word and the context.

25.5.3 Masked language models

A weakness of standard language models such as n-gram models is that the contextualization of each word is based only on the previous words of the sentence. Predictions are made from left to right. But sometimes context from later in a sentence—for example, *feet* in the phrase *rose five feet*—helps to clarify earlier words.

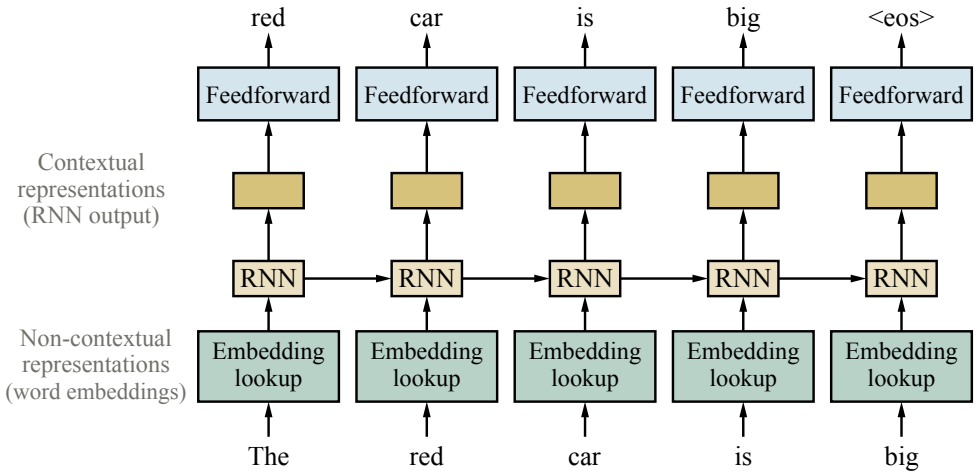

Figure 25.11 Training contextual representations using a left-to-right language model.

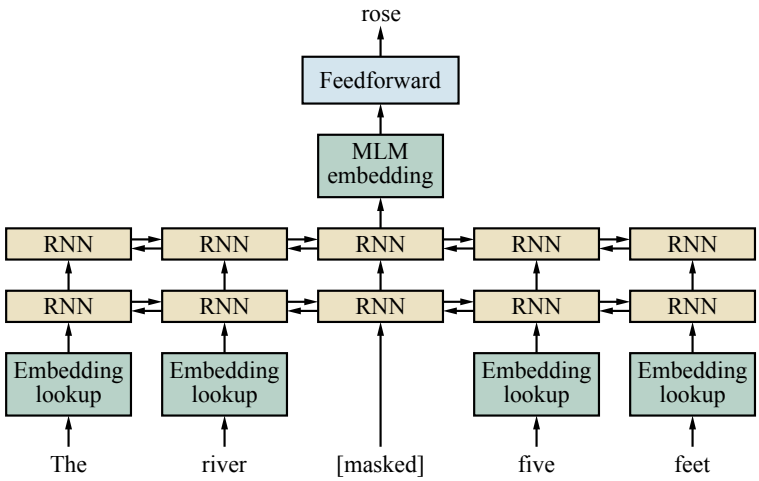

Figure 25.12 Masked language modeling: pretrain a bidirectional model—for example, a multilayer RNN—by masking input words and predicting only those masked words.

One straightforward workaround is to train a separate right-to-left language model that contextualizes each word based on subsequent words in the sentence, and then concatenate the left-to-right and right-to-left representations. However, such a model fails to combine evidence from both directions.

Masked language model (MLM)

Instead, we can use a **masked language model (MLM)**. MLMs are trained by masking (hiding) individual words in the input and asking the model to predict the masked words. For this task, one can use a deep bidirectional RNN or transformer on top of the masked sentence. For example, given the input sentence "*The river rose five feet*" we can mask the middle word to get "*The river __ five feet*" and ask the model to fill in the blank.

1. **What will best separate a mixture of iron filings and black pepper?**
   (a) magnet (b) filter paper (c) triple beam balance (d) voltmeter
2. **Which form of energy is produced when a rubber band vibrates?**
   (a) chemical (b) light (c) electrical (d) sound
3. **Because copper is a metal, it is**
   (a) liquid at room temperature (b) nonreactive with other substances
   (c) a poor conductor of electricity (d) a good conductor of heat
4. **Which process in an apple tree primarily results from cell division?**
   (a) growth (b) photosynthesis (c) gas exchange (d) waste removal

Figure 25.13 Questions from an 8th grade science exam that the ARISTO system can answer correctly using an ensemble of methods, with the most influential being a ROBERTA language model. Answering these questions requires knowledge about natural language, the structure of multiple-choice tests, commonsense, and science.

The final hidden vectors that correspond to the masked tokens are then used to predict the words that were masked—in this example, *rose*. During training a single sentence can be used multiple times with different words masked out. The beauty of this approach is that it requires no labeled data; the sentence provides its own label for the masked word. If this model is trained on a large corpus of text, it generates pretrained representations that perform well across a wide variety of NLP tasks (machine translation, question answering, summarization, grammaticality judgments, and others).

25.6 State of the art

Deep learning and transfer learning have markedly advanced the state of the art for NLP—so much so that one commentator in 2018 declared that "NLP's ImageNet moment has arrived" (Ruder, 2018). The implication is that just as a turning point occurred in 2012 for computer vision when deep learning systems produced surprising good results in the ImageNet competition, a turning point occurred in 2018 for NLP. The principal impetus for this turning point was the finding that transfer learning works well for natural language problems: a general language model can be downloaded and fine-tuned for a specific task.

It started with simple word embeddings from systems such as WORD2VEC in 2013 and GloVe in 2014. Researchers can download such a model or train their own relatively quickly without access to supercomputers. Pretrained contextual representations, on the other hand, are orders of magnitude more expensive to train.

These models became feasible only after hardware advances (GPUs and TPUs) became widespread, and in this case researchers were grateful to be able to download models rather than having to spend the resources to train their own. The transformer model allowed for efficient training of much larger and deeper neural networks than was previously possible (this time due to software advances, not hardware). Since 2018, new NLP projects typically start with a pretrained transformer model.

Although these transformer models were trained to predict the next word in a text, they do a surprisingly good job at other language tasks. A ROBERTA model with some fine-tuning

It is a truth universally acknowledged that the earth is not the center of the universe. There are those who assert there is. I do not accept them, but others I consider to be of the same opinion. The truth is, however, that if there are other than the center, and if there are any other living things in the universe and if they are not human, then we do not yet have our answers. We have to go on. This page gives a simplified, simplified answer to the problem. We don't have all the answers. The truth is, however, that the truth is out there.

When Gregor Samsa woke up one morning, he did not notice anything strange. "When my wife is looking at me, I feel like she is looking at a piece of art," he said. "I think she is admiring something I have created." The idea is that by looking at your own life, you learn something important and become a better person. It is a theory that emerged from psychologist Daniel Goleman's work, in which he asked "How do you know you're not a loser?"

Alice was beginning to get very tired of sitting with her sister on the bank. She sat up, yawned, and said, with a loud little scream, "I hope you don't mind if I keep on doing what I should like to do, and if someone asks me which of us will do more, don't tell them that I won't do much, my dear sister."

All happy families are alike; each happy family is like a garden of paradise. The only difference between happy families and unhappy families, is that the unhappy family doesn't have any flowers or trees.

Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Please fill out the following details. Thank you... Thank you for your interest in this interview. Please wait...

Figure 25.14 Example completion texts generated by the GPT-2 language model, given the prompts in **bold**. Most of the texts are quite fluent English, at least locally. The final example demonstrates that sometimes the model just breaks down.

achieves state-of-the-art results in question answering and reading comprehension tests (Liu *et al.*, 2019b). GPT-2, a transformer-like language model with 1.5 billion parameters trained on 40GB of Internet text, achieves good results on such diverse tasks as translation between French and English, finding referents of long-distance dependencies, and general-knowledge question answering, all without fine-tuning for the particular task. As Figure 25.14 illustrates, GPT-2 can generate fairly convincing text given just a few words as a prompt.

As an example state-of-the-art NLP system, ARISTO (Clark *et al.*, 2019) achieved a score of 91.6% on an 8th grade multiple-choice science exam (see Figure 25.13). ARISTO consists of an ensemble of solvers: some use information retrieval (similar to a web search engine), some do textual entailment and qualitative reasoning, and some use large transformer language models. It turns out that ROBERTA, by itself, scores 88.2% on the test. ARISTO also scores 83% on the more advanced 12th grade exam. (A score of 65% is considered "meeting the standards" and 85% is "meeting the standards with distinction".)

There are limitations of ARISTO. It deals only with multiple-choice questions, not essay questions, and it can neither read nor generate diagrams.[1]

T5 (the Text-to-Text Transfer Transformer) is designed to produce textual responses to various kinds of textual input. It includes a standard encoder–decoder transformer model, pretrained on 35 billion words from the 750 GB Colossal Clean Crawled Corpus (C4). This unlabeled training is designed to give the model generalizable linguistic knowledge that will be useful for multiple specific tasks. T5 is then trained for each task with input consisting of the task name, followed by a colon and some content. For example, when given "translate English to German: *That is good*," it produces as output "*Das ist gut*." For some tasks, the input is marked up; for example in the Winograd Schema Challenge, the input highlights a pronoun with an ambiguous referent. Given the input "referent: *The city councilmen refused the demonstrators a permit because they feared violence*," the correct response is "*The city councilmen*" (not "*the demonstrators*").

Much work remains to be done to improve NLP systems. One issue is that transformer models rely on only a narrow context, limited to a few hundred words. Some experimental approaches are trying to extend that context; the Reformer system (Kitaev *et al.*, 2020) can handle context of up to a million words.

Recent results have shown that using more training data results in better models—for example, ROBERTA achieved state-of-the-art results after training on 2.2 trillion words. If using more textual data is better, what would happen if we included other types of data: structured databases, numerical data, images, and video? We would need a breakthrough in hardware processing speeds to train on a large corpus of video, and we may need several breakthroughs in AI as well.

The curious reader may wonder why we learned about grammars, parsing, and semantic interpretation in the previous chapter, only to discard those notions in favor of purely data-driven models in this chapter? At present, the answer is simply that the data-driven models are easier to develop and maintain, and score better on standard benchmarks, compared to the hand-built systems that can be constructed using a reasonable amount of human effort with the approaches described in Chapter 24. It may be that transformer models and their relatives are learning latent representations that capture the same basic ideas as grammars and semantic information, or it may be that something entirely different is happening within these enormous models; we simply don't know. We do know that a system that is trained with textual data is easier to maintain and to adapt to new domains and new natural languages than a system that relies on hand-crafted features.

It may also be the case that future breakthroughs in explicit grammatical and semantic modeling will cause the pendulum to swing back. Perhaps more likely is the emergence of hybrid approaches that combine the best concepts from both chapters. For example, Kitaev and Klein (2018) used an attention mechanism to improve a traditional constituency parser, achieving the best result ever recorded on the Penn Treebank test set. Similarly, Ringgaard *et al.* (2017) demonstrate how a dependency parser can be improved with word embeddings and a recurrent neural network. Their system, SLING, parses directly into a semantic frame representation, mitigating the problem of errors building up in a traditional pipeline system.

[1] It has been pointed out that in some multiple-choice exams, it is possible to get a good score even without looking at the questions, because there are tell-tale signs in the incorrect answers (Gururangan *et al.*, 2018). That seems to be true for visual question answering as well (Chao *et al.*, 2018) .

There is certainly room for improvement: not only do NLP systems still lag human performance on many tasks, but they do so after processing thousands of times more text than any human could read in a lifetime. This suggests that there is plenty of scope for new insights from linguists, psychologists, and NLP researchers.

Summary

The key points of this chapter are as follows:

- Continuous representations of words with word embeddings are more robust than discrete atomic representations, and can be pretrained using unlabeled text data.
- Recurrent neural networks can effectively model local and long-distance context by retaining relevant information in their hidden-state vectors.
- Sequence-to-sequence models can be used for machine translation and text generation problems.
- Transformer models use self-attention and can model long-distance context as well as local context. They can make effective use of hardware matrix multiplication.
- Transfer learning that includes pretrained contextual word embeddings allows models to be developed from very large unlabeled corpora and applied to a range of tasks. Models that are pretrained to predict missing words can handle other tasks such as question answering and textual entailment, after fine-tuning for the target domain.

Bibliographical and Historical Notes

The distribution of words and phrases in natural language follow **Zipf's Law** (Zipf, 1935, 1949): the frequency of the nth most popular word is roughly inversely proportional to n. That means we have a data sparsity problem: even with billions of words of training data, we are constantly running into novel words and phrases that were not seen before.

Generalization to novel words and phrases is aided by representations that capture the basic insight that words with similar meanings appear in similar contexts. Deerwester *et al.* (1990) projected words into low-dimensional vectors by decomposing the co-occurrence matrix formed by words and the documents the words appear in. Another possibility is to treat the surrounding words—say, a 5-word window—as context. Brown *et al.* (1992) grouped words into hierarchical clusters according to the bigram context of words; this has proven to be effective for tasks such as named entity recognition (Turian *et al.*, 2010). The WORD2VEC system (Mikolov *et al.*, 2013) was the first significant demonstration of the advantages of word embeddings obtained from training neural networks. The GloVe word embedding vectors (Pennington *et al.*, 2014) were obtained by operating directly on a word co-occurrence matrix obtained from billions of words of text. Levy and Goldberg (2014) explain why and how these word embeddings are able to capture linguistic regularities.

Bengio *et al.* (2003) pioneered the use of neural networks for language models, proposing to combine "(1) a distributed representation for each word along with (2) the probability function for word sequences, expressed in terms of these representations." Mikolov *et al.* (2010) demonstrated the use of RNNs for modeling local context in language models. Jozefowicz

et al. (2016) showed how an RNN trained on a billion words can outperform carefully hand-crafted n-gram models. Contextual representations for words were emphasized by Peters *et al.* (2018), who called them ELMo (Embeddings from Language Models) representations.

Perplexity

Note that some authors compare language models by measuring their **perplexity**. The perplexity of a probability distribution is 2^H, where H is the entropy of the distribution (see Section 19.3.3). A language model with lower perplexity is, all other things being equal, a better model. But in practice, all other things are rarely equal. Therefore it is more informative to measure performance on a real task rather than relying on perplexity.

Howard and Ruder (2018) describe the ULMFiT (Universal Language Model Fine-tuning) framework, which makes it easier to fine-tune a pretrained language model without requiring a vast corpus of target-domain documents. Ruder *et al.* (2019) give a tutorial on transfer learning for NLP.

Mikolov *et al.* (2010) introduced the idea of using RNNs for NLP, and Sutskever *et al.* (2015) introduced the idea of sequence to sequence learning with deep networks. Zhu *et al.* (2017) and (Liu *et al.*, 2018b) showed that an unsupervised approach works, and makes data collection much easier. It was soon found that these kinds of models could perform surprisingly well at a variety of tasks, for example, image captioning (Karpathy and Fei-Fei, 2015; Vinyals *et al.*, 2017b).

Devlin *et al.* (2018) showed that transformer models pretrained with the masked language modeling objective can be directly used for multiple tasks. The model was called BERT (Bidirectional Encoder Representations from Transformers). Pretrained BERT models can be fine-tuned for particular domains and particular tasks, including question answering, named entity recognition, text classification, sentiment analysis, and natural language inference.

The XLNet system (Yang *et al.*, 2019) improves on BERT by eliminating a discrepancy between the pretraining and fine-tuning. The ERNIE 2.0 framework (Sun *et al.*, 2019) extracts more from the training data by considering sentence order and the presence of named entities, rather than just co-occurrence of words, and was shown to outperform BERT and XLNet. In response, researchers revisited and improved on BERT: the RoBERTa system (Liu *et al.*, 2019b) used more data and different hyperparameters and training procedures, and found that it could match XLNet. The Reformer system (Kitaev *et al.*, 2020) extends the range of the context that can be considered all the way up to a million words. Meanwhile, ALBERT (A Lite BERT) went in the other direction, reducing the number of parameters from 108 million to 12 million (so as to fit on mobile devices) while maintaining high accuracy.

The XLM system (Lample and Conneau, 2019) is a transformer model with training data from multiple languages. This is useful for machine translation, but also provides more robust representations for monolingual tasks. Two other important systems, GPT-2 (Radford *et al.*, 2019) and T5 (Raffel *et al.*, 2019), were described in the chapter. The later paper also introduced the 35 billion word Colossal Clean Crawled Corpus (C4).

Various promising improvements on pretraining algorithms have been proposed (Yang *et al.*, 2019; Liu *et al.*, 2019b). Pretrained contextual models are described by Peters *et al.* (2018) and Dai and Le (2016).

The GLUE (General Language Understanding Evaluation) benchmark, a collection of tasks and tools for evaluating NLP systems, was introduced by Wang *et al.* (2018a). Tasks include question answering, sentiment analysis, textual entailment, translation, and parsing. Transformer models have so dominated the leaderboard (the human baseline is way down at

ninth place) that a new version, SUPERGLUE (Wang *et al.*, 2019), was introduced with tasks that are designed to be harder for computers, but still easy for humans.

At the end of 2019, T5 was the overall leader with a score of 89.3, just half a point below the human baseline of 89.8. On three of the ten tasks, T5 actually exceeds human performance: yes/no question answering (such as "Is France the same time zone as the UK?") and two reading comprehension tasks involving answering questions after reading either a paragraph or a news article.

Machine translation is a major application of language models. In 1933, Petr Troyanskii received a patent for a "translating machine," but there were no computers available to implement his ideas. In 1947, Warren Weaver, drawing on work in cryptography and information theory, wrote to Norbert Wiener: "When I look at an article in Russian, I say: 'This is really written in English, but it has been coded in strange symbols. I will now proceed to decode.'" The community proceeded to try to decode in this way, but they didn't have sufficient data and computing resources to make the approach practical.

In the 1970s that began to change, and the SYSTRAN system (Toma, 1977) was the first commercially successful machine translation system. SYSTRAN relied on lexical and grammatical rules hand-crafted by linguists as well as on training data. In the 1980s, the community embraced purely statistical models based on frequency of words and phrases (Brown *et al.*, 1988; Koehn, 2009). Once training sets reached billions or trillions of tokens (Brants *et al.*, 2007), this yielded systems that produced comprehensible but not fluent results (Och and Ney, 2004; Zollmann *et al.*, 2008). Och and Ney (2002) show how discriminative training led to an advance in machine translation in the early 2000s.

Sutskever *et al.* (2015) first showed that it is possible to learn an end-to-end sequence-to-sequence neural model for machine translation. Bahdanau *et al.* (2015) demonstrated the advantage of a model that jointly learns to align sentences in the source and target language and to translate between the languages. Vaswani *et al.* (2018) showed that neural machine translation systems can further be improved by replacing LSTMs with transformer architectures, which use the attention mechanism to capture context. These neural translation systems quickly overtook statistical phrase-based methods, and the transformer architecture soon spread to other NLP tasks.

Research on **question answering** was facilitated by the creation of SQUAD, the first large-scale data set for training and testing question-answering systems (Rajpurkar *et al.*, 2016). Since then, a number of deep learning models have been developed for this task (Seo *et al.*, 2017; Keskar *et al.*, 2019). The ARISTO system (Clark *et al.*, 2019) uses deep learning in conjunction with an ensemble of other tactics. Since 2018, the majority of question-answering models use pretrained language representations, leading to a noticeable improvement over earlier systems.

Natural language inference is the task of judging whether a hypothesis (*dogs need to eat*) is entailed by a premise (*all animals need to eat*). This task was popularized by the PASCAL Challenge (Dagan *et al.*, 2005). Large-scale data sets are now available (Bowman *et al.*, 2015; Williams *et al.*, 2018). Systems based on pretrained models such as ELMO and BERT currently provide the best performance on language inference tasks.

The Conference on Computational Natural Language Learning (CoNLL) focuses on learning for NLP. All the conferences and journals mentioned in Chapter 24 now include papers on deep learning, which now has a dominant position in the field of NLP.