Tong Xiao

Jingbo Zhu

Natural Language Processing

Neural Networks and Large Language Models

NATURAL LANGUAGE PROCESSING LAB
NORTHEASTERN UNIVERSITY

&

NIUTRANS RESEARCH

NATURAL LANGUAGE PROCESSING LAB, NORTHEASTERN UNIVERSITY
&
NIUTRANS RESEARCH

June 6, 2025

Contents

Chapter 4

Recurrent and Convolutional Sequence Models

The whole is more than the sum of its parts.

–Aristotle, 384-322 BC [Ross, 1924]

Aristotle might or might not think of linguistic phenomena when having this thought, but it is indeed something we want to express in this chapter: there is something different in a sentence or phrase besides words. Of course, words have meanings, alone. However, when they come together to form a sentence or phrase, the meaning of the whole could be much more complex and diverse. This leads to the most beautiful aspect of language that human beings can express any meaning using a finite set of elements (e.g., words or characters).

The infinite and non-compositional nature of language makes it more difficult to model a sequence of words than to model individual words. A difficulty is that a word may repeatedly alter its meaning in different contexts. Taking the idea of word embedding that a word can be represented as a low-dimensional, real-valued vector, the "meaning" of a language unit could be continuous. It is therefore possible to extend methods of distributed representation from words to sequences of words. This leads us to explore models in which the process of dealing with variable-length word sequences is fundamentally continuous.

Here we consider the general approach to learning the distributed representation of word sequences. In particular, we consider recurrent and convolutional neural networks which have been extensively used in many fields ranging from speech processing to computer vision. For natural language inputs, the result of applying these models is a sequence-level representation of the input. The representation could be either a single real-valued vector, or a sequence of such vectors, each corresponding to a contextualized representation for an input word of the input sequence. Such a model of representation, that can broadly be called an **encoder**, is generally used with a variety of systems whose input is sequential data. We will see several examples of it in this chapter.

4.1 Problem Statement

For many NLP applications, our objective is to make predictions based on an input sequence. Let us consider again the text classification problem mentioned in Chapter 1. If we obtain a text that may talk about food or not, we want to assign one of the two classes to it (say Food or $\mathrm{Not\text{-}food}$). To do this, a common method of classification is to represent the text as a bag of features, denoted as $\mathbf{H}$. Then, a probability is assigned to each of the classes using a probabilistic model $\Pr(y|\mathbf{H})$. The predicted class is the one that has the maximum probability $\hat{y} = \arg\max_y \Pr(y|\mathbf{H})$.

While this is a standard procedure for classification, the underlying idea can be used to describe a general problem. Formally, let $\mathbf{w} = w_1 ... w_m$ be a sequence of words[1]. A sequence-level NLP system can be formulated as a function that maps the sequence $\mathbf{w}$ to some output $\mathbf{y}$. This can be divided into two steps, called the representation (or encoding) step and the prediction step.

- **Representation** (or **Encoding**). It transforms the input sequence $\mathbf{w}$ to some "features" $\mathbf{H}$ by using an encoder $\mathrm{Enc}(\cdot)$:

$$\mathbf{H} \quad = \quad \mathrm{Enc}(\mathbf{w}) \tag{4.1}$$

- **Prediction**. A predictor $\mathrm{Predict}(\cdot)$ takes $\mathbf{H}$ and generates an output:

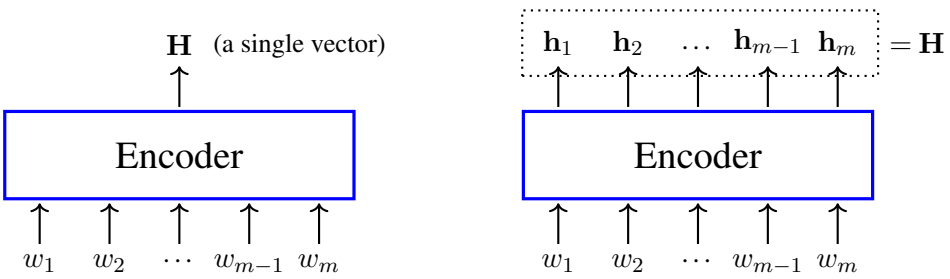
$$\mathbf{y} \quad = \quad \mathrm{Predict}(\mathbf{H}) \tag{4.2}$$

A simple form of $\mathbf{H}$ is a feature vector. For example, $\mathbf{H}$ could be a set of human-designed indicator features extracted from $\mathbf{w}$ (as a high-dimensional sparse representation), or a set of real numbers indicating some latent features (as a low-dimensional dense representation). In NLP, another common form of $\mathbf{H}$ is a sequence of vectors in which each vector $\mathbf{h}_i$ corresponds to an input word w_i (see Figure 4.1). In this case, $\mathbf{h}_i$ can be viewed as a "new" representation of both w_i and its context in $\mathbf{w}$[2]. The correspondence between $\mathbf{h}_i$ and w_i enables the representation to make distinctions among different positions of the sequence, and more importantly, to vary its modeling power for variable-length inputs.

The form of $\mathbf{y}$ is dependent on the problem we intend to deal with. For example, for classification problems, $\mathbf{y}$ is the index of a class (or a distribution of classes); for regression problems, $\mathbf{y}$ is a real number; for translation problems, $\mathbf{y}$ is a sequence of words in another language, and so on. Note that, in the above model, representation and prediction can be regarded as two separate problems. A great advantage of isolating representation and prediction is that we can use the same encoder in many applications with different predictors. This also motivates a promising line of research in which a general-purpose encoder is trained on large-scale data and then used as components in different downstream systems [Peters et al., 2018;

[1] Although we restrict ourselves to word sequences for discussion, the methods can be used to deal with sequences of any language units, e.g., sub-words, characters, etc.

[2] This architecture can be extended to encoders in which the input and output have different lengths, say, the input is $w_1 ... w_m$ and the output is $\mathbf{h}_1 ... \mathbf{h}_n$ ($m \neq n$).



(a) Encoding the sequence as a vector (b) Encoding the sequence as a vector sequence

Figure 4.1: Representing a word sequence as (a) a vector or (b) a sequence of vectors.

Devlin et al., 2019].

There are many possible forms for $\mathrm{Enc}(\cdot)$ and $\mathrm{Predict}(\cdot)$. For text classification, for example, one way is to define $\mathrm{Enc}(\cdot)$ as a function computing a feature vector using a set of hand-crafted feature templates, and define $\mathrm{Predict}(\cdot)$ as a statistical classification model (such as SVMs and maximum entropy-based models). Another way is to define $\mathrm{Enc}(\cdot)$ as a multi-layer neural network that outputs a real-valued vector, and define $\mathrm{Predict}(\cdot)$ as a simple neural network that involves only one Softmax layer. In this chapter we will focus on neural network-based encoders. We will show that such a type of encoder could be applied to a number of NLP tasks in Section 4.5.

4.2 Recurrent Models

A study of various sequence models is not easy work. It is convenient, however, to first introduce one of the most common and practical neural models, called recurrent neural networks (RNNs). We will see later that RNNs are extensively used in sequence modeling, and the techniques presented here are generic and applicable to many systems.

4.2.1 An RNN-based Language Model

Perhaps the most popular use of sequence models in NLP is estimating the probability of a word sequence, also known as language modeling. Mathematically, language modeling is an instance of a well-known problem in the field of **stochastic processes** (or **random processes**): the problem of modeling **time series** data [Hamilton, 1994; Chatfield, 2003; Fuller, 2009]. As a time series, a sequence of words can be treated as a sequence of data points at time intervals that are equally spaced. In this sense, the methods we present here are somewhat general, although the discussion on a broader range of time series problems is beyond the scope of this book.

Given a sequence of words $w_1...w_m$, the goal of language modeling is to compute $\Pr(w_1,...,w_m)$. This joint probability is typically written as a product of conditional probabili-

ties using the chain rule:

$$\Pr(w_1,...,w_m) = \Pr(w_1)\cdot\Pr(w_2|w_1)\cdots\Pr(w_m|w_1,...,w_{m-1}) \tag{4.3}$$

In other words, the problem of generating $w_1...w_m$ is the same as the problem of generating a word w_{i+1} at a time based on the previous words $w_1...w_i$. RNN-based language models represent $w_1...w_i$ via a recurrent unit $\mathrm{RNN}(\cdot)$ [Mikolov et al., 2010], like this

$$\mathbf{h}_i = \mathrm{RNN}(\mathbf{h}_{i-1},\mathbf{x}_i) \tag{4.4}$$

where $\mathbf{x}_i \in \mathbb{R}^{d_e}$ is the word vector (or word embedding) for w_i. Let V be the vocabulary from which we can choose a word. If $w_i \in \mathbb{R}^{|V|}$ is a one-hot word representation [3], $\mathbf{x}_i$ is given by multiplying w_i with the word embedding table $\mathbf{C} \in \mathbb{R}^{|V|\times d_e}$:

$$\begin{aligned}\mathbf{x}_i &= \mathrm{Embed}(w_i)\\ &= w_i\mathbf{C}\end{aligned} \tag{4.5}$$

As shown in Chapter 1, the use of $\mathbf{C}$ transforms a $|V|$-dimensional (and probably high-dimensional) vector to a d_e-dimensional (and probably low-dimensional) vector. Note that $\mathbf{C}$ is essentially a lookup table, with a distinct table entry (i.e., a row) for each word in V. So, the right-hand side of Eq. (4.5) is in practice a function that selects a row from $\mathbf{C}$ with the word index.

Now we go back to Eq. (4.4). The equation is not difficult to understand: the state of the context we have seen so far (i.e., $\mathbf{h}_i$) is some representation of the combination of the current input (i.e., $\mathbf{x}_i$) and the state of the earlier context ($\mathbf{h}_{i-1}$). Put another way, it can be thought of as a process of repeatedly adding information of a new word to a cache of "history". An elegant aspect of this process is that it can be easily implemented by running Eq. (4.4) a number of times until the end of the sequence.

$\mathrm{RNN}(\cdot)$ can be any function that takes $\mathbf{h}_{i-1}$ and $\mathbf{x}_i$, and produces a new vector $\mathbf{h}_i$. The vanilla RNN has a form

$$\mathrm{RNN}(\mathbf{h}_{i-1},\mathbf{x}_i) = \psi(\mathbf{h}_{i-1}\mathbf{U}+\mathbf{x}_i\mathbf{V}) \tag{4.6}$$

where $\psi(\cdot)$ is an activation function, such as $\mathrm{TanH}(\cdot)$ and $\mathrm{Sigmoid}(\cdot)$. Together with Eqs. (4.4) and (4.5), we can define $\mathbf{h}_i$ as a function of $\mathbf{h}_{i-1}$ and w_i

$$\mathbf{h}_i = \psi(\mathbf{h}_{i-1}\mathbf{U}+w_i\mathbf{C}\mathbf{V}) \tag{4.7}$$

where $\mathbf{U} \in \mathbb{R}^{d_h\times d_h}$, $\mathbf{V} \in \mathbb{R}^{d_e\times d_h}$, and $\mathbf{C} \in \mathbb{R}^{|V|\times d_e}$ are learnable parameters of the model, and d_h is a hyper-parameter indicating the number of dimensions of $\mathbf{h}_i$ and $\mathbf{h}_{i-1}$.

We now have an encoder that represents the word sequence $w_1...w_m$ as a sequence of

[3] The one-hot representation w_i is a $|V|$-dimensional vector in which only one entry is 1 and all other entries are zeros. Following the notation used throughout this book, a vector is in general represented as a variable in bold text. Here we treat w_i as a word index and interchangeably use it with the one-hot representation.

RNN's outputs $\mathbf{H} = \{\mathbf{h}_1, ..., \mathbf{h}_m\}$. Given that each $\mathbf{h}_i$ encodes the sub-sequence spanning from w_1 to w_i, we can place a Softmax layer on $\mathbf{h}_i$ to obtain a distribution of words:

$$\mathbf{y}_{i+1} = \text{Softmax}(\mathbf{h}_i\mathbf{O} + \mathbf{b}) \tag{4.8}$$

where $\mathbf{O} \in \mathbb{R}^{d_h \times |V|}$ and $\mathbf{b} \in \mathbb{R}^{|V|}$. Taking the word index w_{i+1}, we have

$$\Pr(w_{i+1}|w_1, ..., w_i) = y_{i+1}(w_{i+1}) \tag{4.9}$$

Thus, we have developed a language model that produces a probability $\Pr(w_{i+1}|w_1, ..., w_i)$ at each step. Figure 4.2 shows an illustration of the RNN-based language model for an example sequence. To run this model on a word sequence, we surely wish to start with predicting w_1 but this requires a preceding word w_0 that is taken as the input. A simple and widely applicable method for giving an appropriate starting state to RNNs is to add a beginning symbol $\langle\text{SOS}\rangle$ to the sequence so that all sequences start with the same "word". Likewise, we can attach an end symbol $\langle\text{EOS}\rangle$ to the sequence to model the completeness of the sequence. This leads to a new form of the probability of the sequence

$$\begin{aligned}\Pr(\langle\text{SOS}\rangle, w_1, ..., w_m, \langle\text{EOS}\rangle) = \; & \Pr(\langle\text{SOS}\rangle) \cdot \\ & \Pr(w_1|\langle\text{SOS}\rangle) \cdot \\ & \Pr(w_2|\langle\text{SOS}\rangle, w_1) \cdot \\ & \cdots \\ & \Pr(w_m|\langle\text{SOS}\rangle, w_1, ..., w_{m-1}) \cdot \\ & \Pr(\langle\text{EOS}\rangle|\langle\text{SOS}\rangle, w_1, ..., w_m)\end{aligned} \tag{4.10}$$

We can simply assume $\Pr(\langle\text{SOS}\rangle) = 1$. To obtain $\Pr(\langle\text{SOS}\rangle, w_1, ..., w_m, \langle\text{EOS}\rangle)$, we take $\langle\text{SOS}\rangle\ w_1...w_m$ as an input sequence and $w_1...w_m\ \langle\text{EOS}\rangle$ as the output sequence.

4.2.2 Training

As a neural network, the RNN-based language model can be trained in a regular way. The training problem has been well discussed in Chapter 2. So, we do not give a full description in this chapter, but a little bit about its basic idea as well as some refinements.

RNN-based language modeling can be framed as a next-step-prediction problem. Suppose we are given a collection of word sequences S. For each sequence $\mathbf{w} = w_1...w_{|\mathbf{w}|}$ in S, we have a sequence of pairs of an input word and the corresponding gold-standard answer, like this[4]

$$\{(w_1, w_2), (w_2, w_3), ..., (w_{|\mathbf{w}|-1}, w_{|\mathbf{w}|})\}$$

The language model takes the input sequence $w_1...w_{|\mathbf{w}|-1}$ and returns a sequence of

[4]While the $\langle\text{SOS}\rangle$ and $\langle\text{EOS}\rangle$ tricks are generally considered in real-world systems, we drop the $\langle\text{SOS}\rangle$ and $\langle\text{EOS}\rangle$ symbols from now on for simplification.

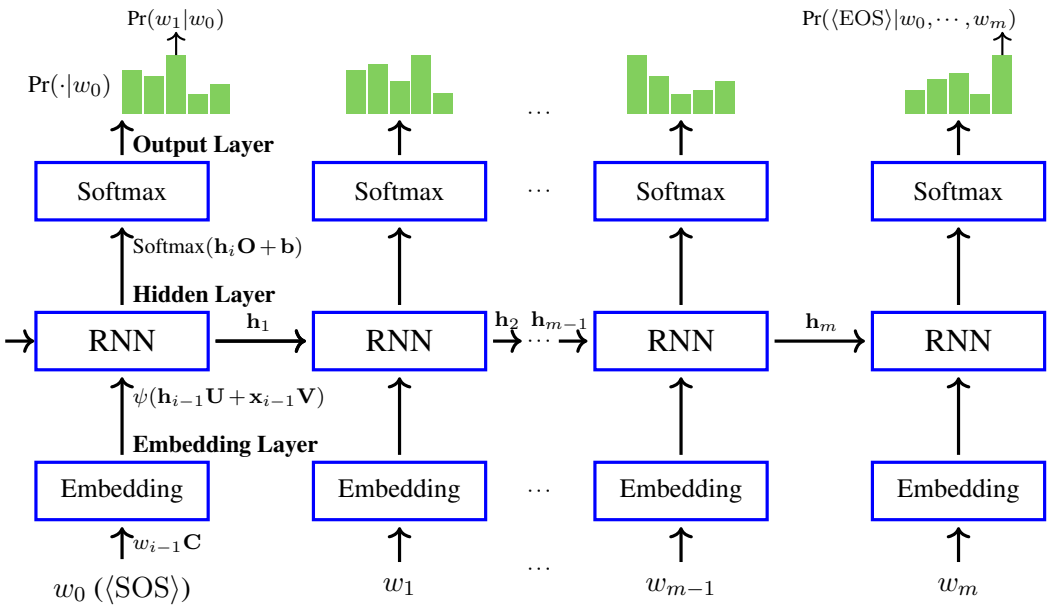


Figure 4.2: Illustration of using an RNN-based language model to calculate $\Pr(\langle\text{SOS}\rangle w_1...w_m\langle\text{EOS}\rangle)$. The input is $\langle\text{SOS}\rangle w_1...w_m$, and the output is the probability $\Pr(w_1|\langle\text{SOS}\rangle)\Pr(w_2|\langle\text{SOS}\rangle\ w_1)...\Pr(\langle\text{EOS}\rangle|\langle\text{SOS}\rangle\ w_1...w_m)$. As $\Pr(\langle\text{SOS}\rangle) = 1$, the probability of generating the sequence is simply $\Pr(\langle\text{SOS}\rangle w_1...w_m\langle\text{EOS}\rangle) = \Pr(\langle\text{SOS}\rangle)\Pr(w_1|\langle\text{SOS}\rangle)\Pr(w_2|\langle\text{SOS}\rangle\ w_1)...\Pr(\langle\text{EOS}\rangle|\langle\text{SOS}\rangle w_1...w_m)$. For each input w_i, we first represent it as a word vector $\mathbf{x}_i$ via the embedding layer, resulting in a sequence of word vectors $\mathbf{x}_0...\mathbf{x}_m$. The RNN layer maps $\mathbf{x}_0...\mathbf{x}_m$ to a sequence of hidden states $\mathbf{h}_1...\mathbf{h}_{m+1}$. In this process, we repeat the same thing: an RNN unit takes both $\mathbf{h}_{i-1}$ and $\mathbf{x}_i$ and produces a new state $\mathbf{h}_i$. On top of that, we use the output layer (Softmax) to obtain $\Pr(w_{i+1}|\langle\text{SOS}\rangle\ w_1...w_i)$.

distributions $\mathbf{y}_2...\mathbf{y}_{|\mathbf{w}|}$. See the following table for an illustration of the inputs and outputs of the model.

Step	History	Input (One-hot)	Output (Distribution)	Gold-Standard
1		w_1	$\mathbf{y}_2$	w_2
2	w_1	w_2	$\mathbf{y}_3$	w_3
3	w_1, w_2	w_3	$\mathbf{y}_4$	w_4
...	...	...	...	...
$\|\mathbf{w}\|-2$	$w_1, w_2, ..., w_{\|\mathbf{w}\|-3}$	$w_{\|\mathbf{w}\|-2}$	$\mathbf{y}_{\|\mathbf{w}\|-1}$	$w_{\|\mathbf{w}\|-1}$
$\|\mathbf{w}\|-1$	$w_1, w_2, ..., w_{\|\mathbf{w}\|-3}, w_{\|\mathbf{w}\|-2}$	$w_{\|\mathbf{w}\|-1}$	$\mathbf{y}_{\|\mathbf{w}\|}$	$w_{\|\mathbf{w}\|}$

A loss function $L(\mathbf{y}_i, w_i)$ is defined to measure how many "errors" we will make if we use $\mathbf{y}_i$ instead of the one-hot representation w_i. A common choice is the cross-entropy loss which computes the divergence of a distribution from another [Mitchell, 1997; Bishop, 2006].

Then, the loss over the entire set is defined to be

$$L = \sum_{\mathbf{w} \in S} \sum_{i=2}^{|\mathbf{w}|} L(\mathbf{y}_i, w_i) \tag{4.11}$$

Once we know the loss, the training of the RNN-based language model can be achieved by using gradient descent. A simple form of this method is the delta rule

$$\theta_{\text{new}} = \theta_{\text{old}} - lr \cdot \frac{\partial L}{\partial \theta} \tag{4.12}$$

where θ stands for the parameters. For the model described in Section 4.2.1, θ includes $\mathbf{C}$, $\mathbf{U}$, $\mathbf{V}$, $\mathbf{O}$ and $\mathbf{b}$. $\frac{\partial L}{\partial \theta}$ is the derivative of the loss with respect to the parameters, called **error gradient**.

Eq. (4.12) can be understood as a process of moving the current parameters a small step in the steepest downhill direction (i.e., the direction of $-\frac{\partial L}{\partial \theta}$). Here lr stands for how far we move in each step of going downhill, also called the learning rate. Obtaining $\frac{\partial L}{\partial \theta}$ often requires a back-propagation process that flushes the error gradient from the output to the input. In modern implementations of deep learning systems, in which neural networks are represented as computation graphs, back-propagation is simple since it is just a by-product of graph traversal and there are many automatic differentiation toolkits to do this. Similar algorithms, called **back-propagation through time** (**BPTT**), were also used in earlier systems [Werbos, 1990]. For further information about training neural networks, see Chapter 2 and/or textbooks on this subject [Goodfellow et al., 2016; Zhang et al., 2021].

If the input is a long sequence, the application of RNNs would result in a deep neural network. In this case, the use of the chain rule of ordered derivatives makes large or small loss derivatives accumulate, and the update to the parameters in Eq. (4.12) is consequently very large or small. These are typically known as the **exploding and vanishing gradient problems**. There are several methods to mitigate these problems for RNNs [Sutskever, 2013]. Some of them are

- **Regularization**. Introducing regularization terms (such as the l_1 and l_2 norms on parameter matrices) into training can avoid models in which most of the parameters have large values, and thus help to avoid exploding gradients. Similarly, one can penalize the cases in which the norms of the gradients are too small [Pascanu et al., 2013].
- **Gradient Clipping**. When the norm of the gradients is too large, it is natural to directly scale down their magnitudes. A simple method is to clip the gradient norm in terms of a threshold τ. If the norm $||\frac{\partial L}{\partial \theta}||$ is larger than τ, we can rescale $\frac{\partial L}{\partial \theta}$ accordingly, say, $\frac{\partial L}{\partial \theta} = \frac{\tau}{||\frac{\partial L}{\partial \theta}||} \cdot \frac{\partial L}{\partial \theta}$.[5]

[5] It is usually formulated as an equation

$$\frac{\partial L}{\partial \theta} = \frac{\tau}{\max(\tau, ||\frac{\partial L}{\partial \theta}||)} \cdot \frac{\partial L}{\partial \theta} \tag{4.13}$$

- **Truncated Back-propagation**. Another idea is to break a long sequence of input-output pairs into shorter pieces, and train RNNs on these separate sub-sequences [Williams and Peng, 1990; Elman, 1990]. This reduces both the cost of training and the risk of too large or small values in accumulating error gradients.
- **Improved Architectures**. It is also possible to redesign the model to overwhelm the limits of standard RNNs, usually using the memory mechanism. In Section 4.3, we will see a few examples of redesigning the recurrent unit for addressing the vanishing gradient problem.
- **Initialization and Constraints of Parameters**. Initializing the model parameters to a desirable region is generally helpful for optimization, and, sometimes, helpful for preventing very small gradients. An alternative method is to randomly set the model and only learn the parameters of the output layers [Jaeger and Haas, 2004].
- **Non-saturating Activations**. Many common activation functions have a compact range of outputs, e.g., the Sigmoid function has a range of $[0, 1]$. They are also called **saturating activation functions**[6]. The use of saturating activation functions often leads to the decay of gradients over layers, i.e., the vanishing gradient. It is therefore promising to use non-saturating activation functions instead, e.g., the ReLU function.
- **Normalization of Activations**. Saturating activations may also result in getting stuck in a saturated region of outputs, and we need a large learning rate to escape from local optimums [Ioffe and Szegedy, 2015]. Thus, the training would be unstable, and subtle changes in inputs and/or model parameters would lead to a big variance in model behavior. A possible solution is to normalize the activations to reduce the variance, e.g., subtracting the mean of the activations in a group of samples (e.g., samples in a mini-batch of training), and dividing by their standard derivation.

4.2.3 Layer Stacking

If we think of the application of a recurrent unit as a function mapping a variable sequence to a new variable sequence of the same length, it is natural to compose this function with another function of the same type, or even with itself. This makes it very easy to extend RNNs to deep neural networks: all you need is to stack RNNs.

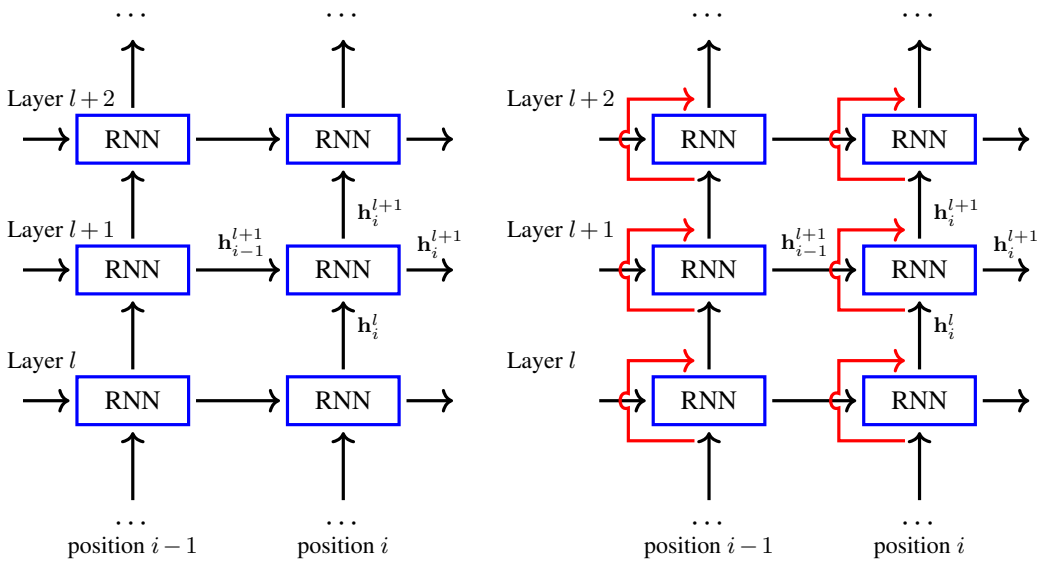
Let $\mathbf{h}_i^l$ be the output of the l-th recurrent unit in the stack at position i. We can apply a new recurrent unit to $\mathbf{h}_i^l$, resulting in a new output at level $l+1$

$$\mathbf{h}_i^{l+1} = \text{RNN}(\mathbf{h}_{i-1}^{l+1}, \mathbf{h}_i^l) \tag{4.14}$$

where $\mathbf{h}_{i-1}^{l+1}$ is the output of the previous step at level $l+1$. To make Eq. (4.14) well-formed, we typically define $\mathbf{h}_i^0 = \mathbf{x}_i$. In other words, the stack starts off with the word vector $\mathbf{x}_i$, then a series of RNN outputs (i.e., $\mathbf{h}_i^1$, $\mathbf{h}_i^2$, $\mathbf{h}_i^3$, etc).

To illustrate, Figure 4.3 (a) shows a stacked RNN for language modeling. We see that

[6] An activation function $f(x)$ is non-saturating if and only if when $x \to \infty$ (or $-\infty$), $f(x) \to \infty$. An activation function is saturating if it is not a non-saturating activation function.



(a) A 3-layer RNN.

(b) A 3-layer RNN with residual connections.

Figure 4.3: 3-layer RNNs (with and without residual connections). To stack RNN layers, we feed the output of layer l to layer $l+1$. Thus the output of layer $l+1$ is given by $\mathbf{h}_i^{l+1} = \mathrm{RNN}(\mathbf{h}_{i-1}^{l+1}, \mathbf{h}_i^l)$. Lines in red color stand for the residual connections which directly add the input of a layer to its output, resulting in $\mathbf{h}_i^{l+1} = \mathrm{RNN}(\mathbf{h}_{i-1}^{l+1}, \mathbf{h}_i^l) + \mathbf{h}_i^l$.

applying a stack of recurrent units is equivalent to creating multiple layers of RNNs simultaneously. However, there would be a risk of confusion if we call an unrolled recurrent network a *layer*, as the term *layer* typically refers to a set of neurons receiving the same inputs in a feed-forward neural network. Here we extend the term *layer* to cover a more general concept: a group of neurons that are topologically placed on the same level. So, we say that the language model in Figure 4.3 has 3 RNN layers.

Stacking multiple layers of RNNs, we build a model which is deeper but more difficult to train. This difficulty arises in part from the barriers of passing information through many-layered RNNs. To make the training easier, a widely-used approach is to introduce **skip connections** or **residual connections** into a multi-layer neural network [He et al., 2016]. These connections are intended to leverage an additional path to allow information to skip layers. As described in Chapter 2, the form of a residual neural network is given by

$$\mathbf{y}^{l+1} = F(\mathbf{y}^l) + \mathbf{y}^l \tag{4.15}$$

where $\mathbf{y}^l$ is the output of layer l. Extending this formulation to Eq. (4.14) leads to multi-layer RNNs with residual connections, given by

$$\mathbf{h}_i^{l+1} = \mathrm{RNN}(\mathbf{h}_{i-1}^{l+1}, \mathbf{h}_i^l) + \mathbf{h}_i^l \tag{4.16}$$

The only difference from Eq. (4.14) is that we introduce the identity map of $\mathbf{h}_i^l$ to the right-hand side of Eq. (4.16). Thus, the input $\mathbf{h}_i^l$ is directly accessible from layer $l+1$. This greatly simplifies the way that the information flows through the neural network, and allows the system to "skip" layers in propagating errors. Figure 4.3 (b) shows a 3-layer RNN with residual connections.

4.2.4 Bi-directional Models

The use of RNNs enables us to formulate the problem of encoding a word sequence as a problem of left-to-right generation of words. One advantage of this approach is that the modeling of context words arises naturally: the output of an RNN unit in some way describes the history words up to that point. This feature makes it very straightforward to model the probability distribution $\Pr(w_{i+1}|w_1,...,w_i)$, as we can use $\mathbf{h}_i$ as a representation of the context $w_1...w_i$, that is, $\Pr(w_{i+1}|w_1,...,w_i) = \Pr(w_{i+1}|\mathbf{h}_i)$.

The left-to-right generation is widely used in sequence generation, such as machine translation. It can be viewed as an instance of **autoregressive processes** (**AR processes**) in which the state of a variable is dependent on the state of the previous variables [Chatfield, 2003; Box et al., 2015][7]. However, such a method is not the only choice for modeling sequences. We do not even necessarily restrict ourselves to language modeling for training a sequence encoder. This gives rise to an interesting question: how can we develop an encoder of word sequences without assumptions regarding the predictor? Answering the question leads us to isolate the learning of the text encoder from a specific NLP task, and to regard it as a separate task whose result can be applied to many other systems. A more detailed discussion is not the focus here and we leave it to subsequent chapters.

We now present a simple extension of the left-to-right sequence model by returning to RNNs. Note that in sequence modeling our desire is some representation of the entire sequence. A problem with usual RNNs is that they are **uni-directional models** in which the context words following w_i are absent. To consider both the left and right contexts of a given word, we can instead use **bi-directional models**. Figure 4.4 shows an example of the bi-directional RNN. There are two sub-models: a left-to-right RNN and a right-to-left RNN. They have the

[7] As a stochastic process, an autoregressive process expresses a variable at time t by relating it to the past values of the process and the current value of an error process [Chatfield, 2003]. Formally, a time series $\{z_1,...,z_T\}$ describes an autoregressive process of order p if for any $t \in \{p+1,...,T\}$

$$z_t = \sum_{i=1}^{p} \alpha_i z_{t-i} + \epsilon_t \tag{4.17}$$

where $\{\alpha_1,...,\alpha_p\}$ are the parameters of the process, and ϵ_t is the error at time t. This process is called *regressive* because it has the same form as the **multiple linear regression model**. The prefix *auto-* comes from the way we regress z_t: z_t is dependent on its past values instead of additional independent variables. One way to interpret language modeling in an autoregressive process perspective is to simply treat $\{z_1,...,z_T\}$ as representations of a sequence of words $\{w_1,...,w_T\}$. Thus, we can gain some idea of predicting w_t using previous words $\{w_1,...,w_t\}$ by considering the autoregressive property of the problem. However, it should be noted that most of the sequence generation models used in NLP are not mathematically equivalent to Eq. (4.17), although they are often called regressive models. For example, the RNN-based language model discussed here is not a linear model. Rather, it takes layers of non-linearity to describe the complex relationships among words.

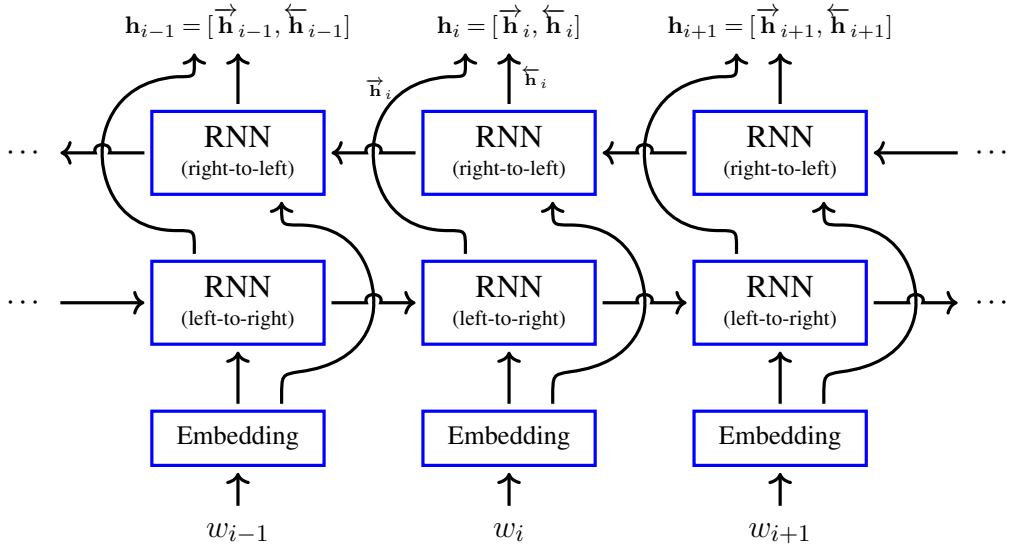


Figure 4.4: A bi-directional RNN model. Given a word sequence, we run an RNN from left to right and another RNN from right to left. Therefore, at each position we obtain a left-to-right representation and a right-to-left representation. The output is the concatenation of the two representations so that it involves both the left and right contexts.

same architecture but work in opposite directions. For each input word w_i, the left-to-right RNN outputs a vector representing the context $\{w_1, ..., w_i\}$ (denoted by $\overrightarrow{\mathbf{h}}_i$), and the right-to-left RNN outputs a vector representing the context $\{w_i, ..., w_m\}$ (denoted by $\overleftarrow{\mathbf{h}}_i$). We can concatenate $\overrightarrow{\mathbf{h}}_i$ and $\overleftarrow{\mathbf{h}}_i$ to obtain a bi-directional representation

$$\mathbf{h}_i = [\overrightarrow{\mathbf{h}}_i, \overleftarrow{\mathbf{h}}_i] \tag{4.18}$$

Thus, the bi-directional RNN has the same form of output as that of the uni-directional RNN, that is, a sequence of vectors $\{\mathbf{h}_1, .., \mathbf{h}_m\}$. Unlike the uni-directional RNN, the representation $\mathbf{h}_i$ here describes the context on both sides.

For a stronger model, the bi-directional RNN can be extended to a neural network of multiple RNN layers. For example, we can run deep RNNs in two directions and combine their results as in Eq. (4.18). Such model architectures have been extensively used in language and speech processing tasks, including machine translation [Wu et al., 2016], sentiment analysis [Tang et al., 2015], POS tagging [Huang et al., 2015], speech recognition [Graves et al., 2013a;b], and so on.

4.3 Memory

RNNs can be appropriate for sequence learning in which we summarize at each step the past inputs and then make some prediction on this summary of the "history". A benefit of RNNs is that we can represent a history of arbitrary length as a fixed-size vector, and update it when

new information arrives. In other words, we have a **memory**, though not explicitly defined, to store the information. Next we show that such a memory mechanism is general and can be used to improve sequence models.

4.3.1 Memory as A System

In psychology, memory is the ability of the mind to retain and recall information. There are many cognitive models of psychology. A well-known model is **the multi-store model** [Atkinson and Shiffrin, 1968]. It defines memory as a system consisting of three components: short-lived **sensory memory**, **short-term memory**, and **long-term memory**. The sensory memory retains the sensory information that is very quickly ceased, such as immediate data from the senses of sight and smell. The short-term memory stores information for a longer time but is not permanent. An example of the short-term memory is that we try to memorize a sequence of digits (e.g., a phone number) but may forget it after a short while. The long-term memory is permanent. This also means that the information is retained indefinitely. For example, adults can remember details of the events that occurred in their childhood.

Given this categorization, there appear to be interesting connections between the above model of memory and the neural networks we discuss here. For example, the state of a recurrent unit can be simply thought of as a short-term memory. It maintains information until we get to the end of a sequence and would be reset if we switch to a new sequence[8]. On the other hand, the entire language model and associated parameters perform more like a long-term memory: the language model is intended to learn and memorize some useful information about probabilistic word prediction from the text, so that it can be used whenever we want to. Moreover, there are other concepts that may stem from psychology but are used in several different fields. For example, **coding** or encoding is referred to as how the information is stored in a memory, **duration** is referred to as how long the information is stored in a memory, and **capacity** is referred to as how much information is stored in a memory.

In machine learning and NLP, we can gain an understanding of memory by considering it from an information processing point of view. Broadly, memory can be viewed as a system that writes information to a "storage" and reads it when queried. It has the following functions.

- **Encoding**. The input of the system is encoded in a form that is easy to process. For text inputs, this can be simply thought of as the same encoding process as we discuss in both Chapter 3 and this chapter: a word or a sequence of words is represented as a feature vector or a sequence of feature vectors.
- **Update**. Given the encoded information, we store it in the memory. This operation is generally dependent on the organization of the memory. For example, one can treat a group of encoded items as a datastore with an indexing system. In this case, storing an item requires finding the right place to keep it. Alternatively, one can represent the memory as a single vector of numbers.
- **Retrieval**. The stored information can be retrieved. This typically involves matching

[8] Another explanation is that the state of a recurrent unit at step i may contain little information about very early steps.

each item in the memory against an input query. If the memory is represented in a simpler form, such as a vector, it may not be explicitly retrieved, and we return the entire memory when required.

These functions can be designed in many different ways, leading to a variety of NLP systems. One simple example is information retrieval [Manning et al., 2008]. A typical information retrieval system indexes a large number of documents (or other resources) and allows users to search for interested information in this collection of documents. To enable search, documents are represented in forms that are convenient to use, for example, we may use the bag-of-words model to compute the matching score between a document and a query, and may use the inverted index to make an efficient mapping of a document to its location in the storage. Systems of this type cover a wide range of applications, including translation memory, dialogue, summarization, document classification, and so on.

Another design choice made for memory systems is to consider, either partially or fully, a continuous form for the above components. One method is to encode each input item as a real-valued vector (e.g., a word embedding) but use the same modules of update and retrieval as in usual information retrieval-like systems [Weston et al., 2015; Khandelwal et al., 2019]. An alternative method is to adopt differentiable functions for all the steps in building and accessing the memory. These models are typically implemented using neural networks and trained using gradient descent [Sukhbaatar et al., 2015; Graves et al., 2014; Kumar et al., 2016; Graves et al., 2016; Miller et al., 2016]. This idea motivates work on exploring approaches to coupling neural networks with memories, such as **end-to-end memory networks** and **neural Turing machines**. Note that the above models are sometimes called **external memories**, as they are used as separate modules working with other systems.

Memory can also work as an internal or hidden component of a system. In this case, the memory is typically rebuilt for each input sample, and so it can be regarded as an instance of the short-term memory. There are various ways of using this type of memory to improve sequence models. In the remainder of the section, we will focus on using the memory mechanism in RNNs. In Chapter 5, we will see how the idea of memory is extended to model the correspondence between tokens of two sequences.

4.3.2 Long Short-Term Memory

In the vanilla RNN presented in Section 4.2.1, the summarization of the context words was given by the output of a recurrent unit. It implicitly defines a memory, and thus enables the prediction based on past information for an arbitrary duration. The memory simply combines the representations of the earlier history $w_1 \ldots w_{i-1}$ and the input at the current step w_i, but does not consider how much information from different steps should be squeezed into a fixed-length representation. A problem with this model is that, if long-term dependencies are required for prediction, memory may provide little information about it, and it may be hard to learn these dependencies through back-propagation [Bengio et al., 1994; Pascanu et al., 2013]. A more powerful approach, therefore, is to compute what should be retained at each step, and to let the model learn to decide whether to memorize or forget.

Long short-term memory (**LSTM**) is perhaps the best-known variant of RNNs to accomplish the above goal [Hochreiter and Schmidhuber, 1997]. The basic idea of LSTM is that a recurrent unit can learn to memorize useful things and forget unuseful things by maintaining an explicit memory [Gers et al., 2000]. To this end, the vanilla recurrent unit is replaced with an LSTM unit that is made up of an output vector (call it a **recurrent cell**), a memory vector (call it a **memory cell**), and three gates to control the information flow inside the LSTM unit. As an extension to RNNs, an LSTM network deals with an input sequence as usual: it starts with some initial states, and then repeatedly takes an input and outputs a vector. A key difference between LSTM networks and RNNs is that the LSTM unit of step i takes both the recurrent cell and the memory cell of its previous step. The form of an LSTM unit is given by

$$(\mathbf{h}_i, \mathbf{c}_i) \quad = \quad \text{LSTM}(\mathbf{h}_{i-1}, \mathbf{c}_{i-1}, \mathbf{x}_i) \tag{4.19}$$

where $\mathbf{h}_i \in \mathbb{R}^{d_h}$ is the recurrent cell of step i, $\mathbf{c}_i \in \mathbb{R}^{d_h}$ is the memory cell of step i, and $\mathbf{x}_i \in \mathbb{R}^{d_e}$ is the input of step i. Given $\text{LSTM}(\cdot)$, applying the LSTM model is straightforward. We simply repeat the call of $\text{LSTM}(\cdot)$ for the inputs $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$ and obtain the outputs $\{\mathbf{h}_1, ..., \mathbf{h}_m\}$. This resembles the way we use vanilla RNNs, making it very easy to extend LSTM to multi-layer models (see Section 4.2.3) and bi-directional models (see Section 4.2.4).

We can divide $\text{LSTM}(\cdot)$ into three steps.

- **Step 1: Forget**. Assuming that $\mathbf{c}_{i-1}$ contains the information that the model memorizes at step $i-1$, we need to determine how much information in $\mathbf{c}_{i-1}$ is discarded in building $\mathbf{c}_i$. To do this, a gate is used to control to what extent we forget for each dimension of $\mathbf{c}_{i-1}$. The **forget gate** is defined to be:

$$\mathbf{f}_i \quad = \quad \text{Sigmoid}(\mathbf{h}_{i-1}\mathbf{U}_\text{f} + \mathbf{x}_i\mathbf{V}_\text{f} + \mathbf{b}_\text{f}) \tag{4.20}$$

  where $\mathbf{f}_i \in [0,1]^{d_h}$ is a vector with the same number of dimensions as $\mathbf{c}_{i-1}$. The Sigmoid function maps the input data to the range $[0,1]$. Thus, an entry of $\mathbf{f}_i$ indicates how much is preserved for the same entry of $\mathbf{c}_{i-1}$. Taking this further, $\mathbf{f}_i \odot \mathbf{c}_{i-1}$ describes the memory that is left out after passing through the forget gate. See Figure 4.5 (a) for an illustration of the forget gate in the LSTM unit.

- **Step 2: Update**. Next we update the memory by considering both the previous state of the memory (i.e., $\mathbf{c}_{i-1}$) and the input of the LSTM unit (i.e., $\mathbf{x}_i$ and $\mathbf{h}_{i-1}$). We first combine $\mathbf{x}_i$ and $\mathbf{h}_{i-1}$ using a simple neural network, like this

$$\hat{\mathbf{c}}_i \quad = \quad \text{TanH}(\mathbf{h}_{i-1}\mathbf{U}_\text{c} + \mathbf{x}_i\mathbf{V}_\text{c} + \mathbf{b}_\text{c}) \tag{4.21}$$

  $\hat{\mathbf{c}}_i$ can be treated as the new information we intend to add to the memory at step i. Again, we need a way to control the amount of information coming into the memory. Hence we define an **input gate** as

$$\mathbf{g}_i \quad = \quad \text{Sigmoid}(\mathbf{h}_{i-1}\mathbf{U}_\text{g} + \mathbf{x}_i\mathbf{V}_\text{g} + \mathbf{b}_\text{g}) \tag{4.22}$$

This equation is similar to Eq. (4.20) but with different parameters. We then define $\mathbf{g}_i \odot \hat{\mathbf{c}}_i$ to be the actual new information that we are interested in. Taking both $\mathbf{f}_i \odot \mathbf{c}_{i-1}$ and $\mathbf{g}_i \odot \hat{\mathbf{c}}_i$, the memory cell at step i is given by

$$\mathbf{c}_i = \mathbf{f}_i \odot \mathbf{c}_{i-1} + \mathbf{g}_i \odot \hat{\mathbf{c}}_i \tag{4.23}$$

In other words, we forget something old in $\mathbf{c}_{i-1}$ and memorize something new in $\hat{\mathbf{c}}_i$. See Figure 4.5 (b) for an illustration of the update step.

- **Step 3: Output**. In the last step we generate the output $\mathbf{h}_i$ based on the memory $\mathbf{c}_i$. Instead of copying $\mathbf{c}_i$ to $\mathbf{h}_i$, we feed $\mathbf{c}_i$ to a hyperbolic function and multiply its result with the output gate. Like Eqs. (4.20) and (4.22), the output gate is given by

$$\mathbf{o}_i = \text{Sigmoid}(\mathbf{h}_{i-1}\mathbf{U}_\text{o} + \mathbf{x}_i\mathbf{V}_\text{o} + \mathbf{b}_\text{o}) \tag{4.24}$$

Then, the output of the LSTM unit is defined to be

$$\mathbf{h}_i = \mathbf{o}_i \odot \text{TanH}(\mathbf{c}_i) \tag{4.25}$$

See Figure 4.5 (c) for an illustration of the output step.

The LSTM model is parameterized by $\mathbf{U}_\text{f}, \mathbf{U}_\text{c}, \mathbf{U}_\text{g}, \mathbf{U}_\text{o} \in \mathbb{R}^{d_h \times d_h}$, $\mathbf{V}_\text{f}, \mathbf{V}_\text{c}, \mathbf{V}_\text{g}, \mathbf{V}_\text{o} \in \mathbb{R}^{d_e \times d_h}$, and $\mathbf{b}_\text{f}, \mathbf{b}_\text{c}, \mathbf{b}_\text{g}, \mathbf{b}_\text{o} \in \mathbb{R}^{d_h}$. Compared with vanilla RNNs, additional parameters are introduced here because of the use of three gates. In practice one can implement them in many different ways, e.g., using activation functions other than $\text{Sigmoid}(\cdot)$ and $\text{TanH}(\cdot)$, removing the bias terms $\mathbf{b}_\text{f}$, $\mathbf{b}_\text{c}$, $\mathbf{b}_\text{g}$, and $\mathbf{b}_\text{o}$, and so on. Training LSTM models follows the standard paradigm of training RNN-based models. For example, we can build an LSTM-based language model and train it by using the methods presented in Section 4.2.2.

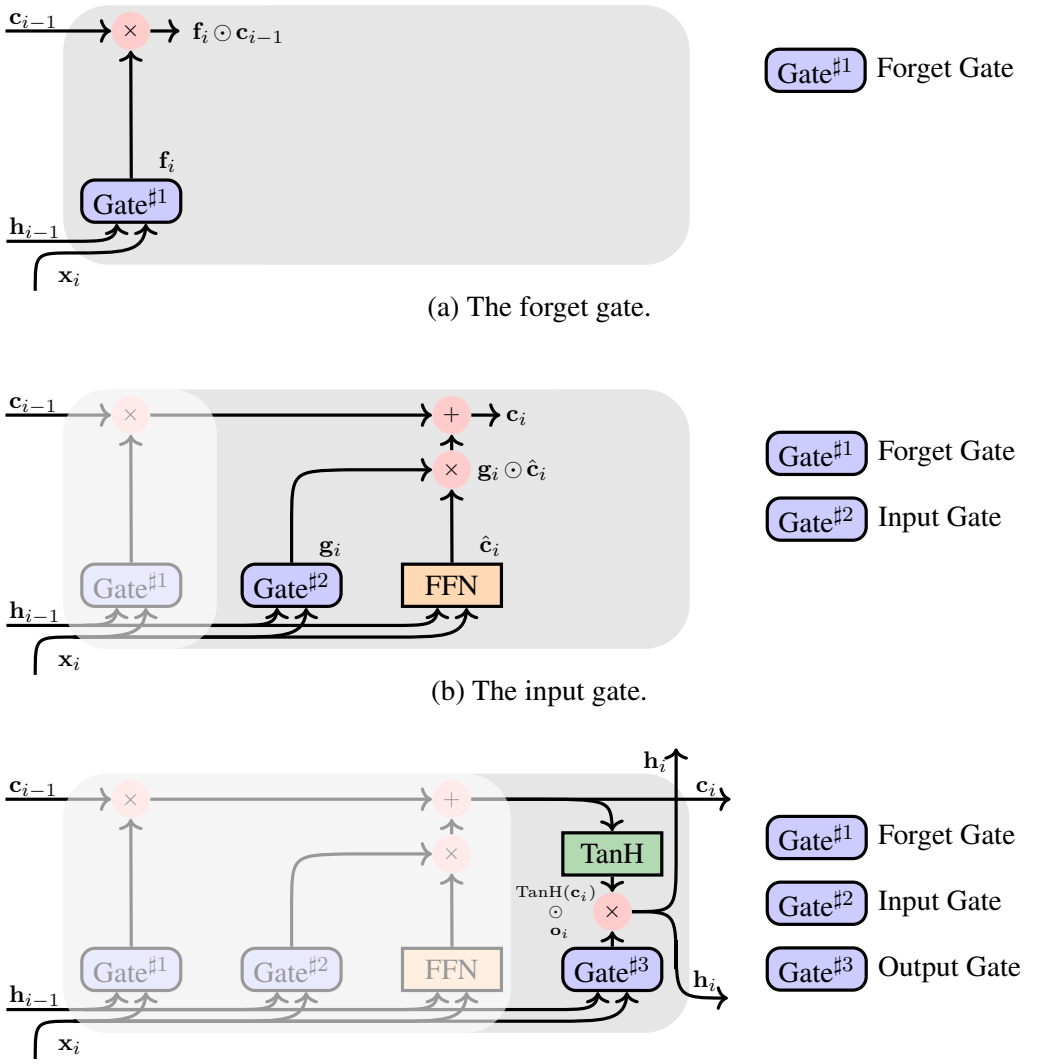
4.3.3 Gated Recurrent Units

Above, we saw the important role played by the gate units and the memory cell. In general the use of these neural networks makes the model computationally more expensive. An alternative to LSTM in a cheap case, namely **gated recurrent units** (**GRUs**), uses a simplified model structure with fewer gate functions [Cho et al., 2014; Chung et al., 2014]. Unlike LSTM, a GRU does not have a memory cell so, as an RNN unit, it takes both the previous state vector $\mathbf{h}_{i-1}$ and the current input vector $\mathbf{x}_i$, and produces the current state vector $\mathbf{h}_i$.

In GRUs, there are two gate units: the **reset gate** and the **update gate**. The reset gate, as the name suggests, is used to reset (or rescale) the state of the GRU (i.e., $\mathbf{h}_{i-1}$). Following the gate functions used in LSTM, the reset gate is defined to be

$$\mathbf{r}_i = \text{Sigmoid}(\mathbf{h}_{i-1}\mathbf{U}_\text{r} + \mathbf{x}_i\mathbf{V}_\text{r} + \mathbf{b}_\text{r}) \tag{4.26}$$

where $\mathbf{r}_i \in [0,1]^{d_h}$ is a vector of scalars, each dimension describing how much information in the corresponding dimension of $\mathbf{h}_{i-1}$ is retained. Thus, we have a representation of retained



(a) The forget gate.

(b) The input gate.

(c) The output gate.

Figure 4.5: The architecture of the LSTM unit. At step i, it takes the input $\mathbf{x}_i$, and then updates both the memory cell ($\mathbf{c}_{i-1} \rightarrow \mathbf{c}_i$) and the recurrent cell ($\mathbf{h}_{i-1} \rightarrow \mathbf{h}_i$). This process involves three gates: the forget gate controls how much information in $\mathbf{c}_{i-1}$ is retained at step i, the input gate controls how much information in $\mathbf{c}_{i-1}$ and $\mathbf{x}_i$ is retained at step i, and the output gate controls how much information in $\mathbf{c}_i$ is used to form $\mathbf{h}_i$.

information

$$\upsilon_{i-1} \quad = \quad \mathbf{r}_i \odot \mathbf{h}_{i-1} \tag{4.27}$$

Taking both the retained information υ_{i-1} and the current input $\mathbf{x}_i$, a new state vector is defined

to be

$$\hat{\mathbf{h}}_i = \text{TanH}(v_{i-1}\mathbf{U}_\text{h} + \mathbf{x}_i\mathbf{V}_\text{h} + \mathbf{b}_\text{h}) \tag{4.28}$$

The update gate is then given by

$$\mathbf{u}_i = \text{Sigmoid}(\mathbf{h}_{i-1}\mathbf{U}_\text{u} + \mathbf{x}_i\mathbf{V}_\text{u} + \mathbf{b}_\text{u}) \tag{4.29}$$

$\mathbf{u}_i$ can be thought of as a coefficient vector which could be used to control the trade-off in choosing the new state vector $\hat{\mathbf{h}}_i$ or the old state vector $\mathbf{h}_{i-1}$. Finally, the output of the GRU is defined as a linear interpolation of $\hat{\mathbf{h}}_i$ and $\mathbf{h}_{i-1}$

$$\mathbf{h}_i = \mathbf{u}_i \odot \hat{\mathbf{h}}_i + (1 - \mathbf{u}_i) \odot \mathbf{h}_{i-1} \tag{4.30}$$

Figure 4.6 shows how the information flows in a GRU unit. The parameters here are $\mathbf{U}_\text{r}, \mathbf{U}_\text{h}, \mathbf{U}_\text{u} \in \mathbb{R}^{d_h \times d_h}$, $\mathbf{V}_\text{r}, \mathbf{V}_\text{h}, \mathbf{V}_\text{u} \in \mathbb{R}^{d_e \times d_h}$, and $\mathbf{b}_\text{r}, \mathbf{b}_\text{h}, \mathbf{b}_\text{u} \in \mathbb{R}^{d_h}$. Therefore, the GRU model is smaller than the LSTM model because of the use of fewer gate units. Note that removing the memory cell makes GRUs more efficient. In this case, the role of memory is implicitly played by GRU's output $\mathbf{h}_i$, and we maintain it by memorizing more "important" information.

4.4 Convolutional Models

In this section we describe another type of model for sequence modeling, called convolutional neural networks (CNNs). Our description is mostly standard, but not a full introduction to the numerous variants of CNNs and cutting-edge techniques. In particular, we focus on using CNNs to deal with sequential data and presenting some refinements.

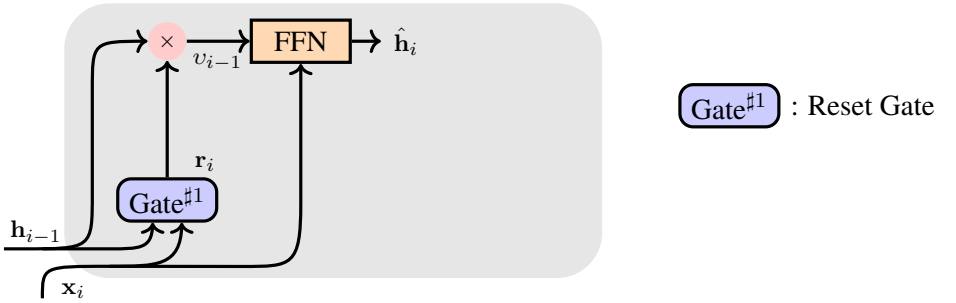
4.4.1 Convolution

CNNs feature their shared-weight architectures by which a kernel or filter slides over the input data and produces a map of features. The idea is that the filter only receives signals from a restricted region of data at a time (call it the **receptive field**), and computes the weighted sum of these input signals. To illustrate this, we follow the convention that a filter in CNNs is generally used to deal with 2D data. Consider a 3×3 data matrix

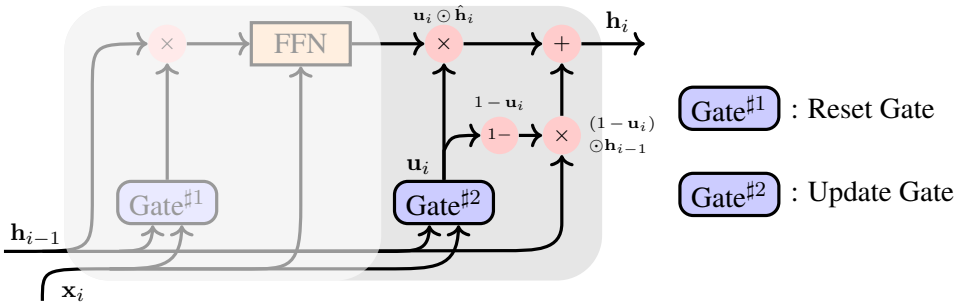
$$\mathbf{A} = \begin{bmatrix} 1 & 9 & 7 \\ 3 & 1 & 2 \\ 0 & 1 & -1 \end{bmatrix} \tag{4.31}$$

and a 2×2 filter with a weight matrix

$$\mathbf{W} = \begin{bmatrix} 2 & 0 \\ 2 & 2 \end{bmatrix} \tag{4.32}$$



(a) The reset gate.



(b) The update gate.

Figure 4.6: The architecture of the GRU. Unlike the LSTM unit, the GRU does not involve a memory cell, and thus follows the same input and output forms of a standard RNN unit. There are two gates in the GRU. The reset gate controls how much information in $\mathbf{h}_{i-1}$ is retained at step i. The retained information is then taken to fuze with the input $\mathbf{x}_i$, generating the candidate output $\hat{\mathbf{h}}_i$. The update gate seeks a balance between $\hat{\mathbf{h}}_i$ and $\mathbf{h}_{i-1}$ in computing the final output of the GRU.

We can apply the filter to every 2×2 sub-matrix of $\mathbf{A}$ (there are four 2×2 sub-matrices here), and compute the sum of the 2×2 entries weighted by $\mathbf{W}$. For example, consider the 2×2 sub-matrix in the upper left corner of $\mathbf{A}$. The output of the filter is given by

$$\begin{aligned} \text{Conv}(\begin{bmatrix} 1 & 9 \\ 3 & 1 \end{bmatrix}, \mathbf{W}) &= \text{Conv}(\begin{bmatrix} 1 & 9 \\ 3 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 0 \\ 2 & 2 \end{bmatrix}) \\ &= 1 \times 2 + 9 \times 0 + 3 \times 2 + 1 \times 2 \\ &= 10 \end{aligned} \tag{4.33}$$

$\text{Conv}(\cdot)$ defines a **convolution operation** that sums the entries of the element-wise product of the two matrices. The convolution operation can be extended to cover the entire input matrix

by sliding the filter over it, as follows

$$\begin{aligned}\text{Conv}(\mathbf{A},\mathbf{W}) &= \text{Conv}(\begin{bmatrix}1 & 9 & 7\\ 3 & 1 & 2\\ 0 & 1 & -1\end{bmatrix},\mathbf{W})\\ &= \begin{bmatrix}\text{Conv}(\begin{bmatrix}1 & 9\\ 3 & 1\end{bmatrix},\mathbf{W}) & \text{Conv}(\begin{bmatrix}9 & 7\\ 1 & 2\end{bmatrix},\mathbf{W})\\ \text{Conv}(\begin{bmatrix}3 & 1\\ 0 & 1\end{bmatrix},\mathbf{W}) & \text{Conv}(\begin{bmatrix}1 & 2\\ 1 & -1\end{bmatrix},\mathbf{W})\end{bmatrix}\\ &= \begin{bmatrix}10 & 24\\ 8 & 2\end{bmatrix}\end{aligned} \tag{4.34}$$

The output $\begin{bmatrix}10 & 24\\ 8 & 2\end{bmatrix}$ is also called the **feature map** for the filter $\mathbf{W}$ on $\mathbf{A}$. Sometimes, the convolution operation $\text{Conv}(\mathbf{A},\mathbf{W})$ is written as $\mathbf{A}*\mathbf{W}$ where the symbol $*$ stands for the **convolution product**.[9]

Now let us consider a more general description of convolution in CNNs. Suppose that $\mathbf{A}$ is a multi-dimensional data array. A filter defines a window (or receptive field) on $\mathbf{A}$. We can move the window on $\mathbf{A}$ in different directions. This results in a set of data arrays, denoted by Ω. Each data array $\mathbf{a}_p \in \Omega$ is formed by the elements from the corresponding region of $\mathbf{A}$. For example, there are four sub-matrices in Eq. (4.34)): $\mathbf{a}_1 = \begin{bmatrix}1 & 9\\ 3 & 1\end{bmatrix}$, $\mathbf{a}_2 = \begin{bmatrix}9 & 7\\ 1 & 2\end{bmatrix}$, $\mathbf{a}_3 = \begin{bmatrix}3 & 1\\ 0 & 1\end{bmatrix}$, and $\mathbf{a}_4 = \begin{bmatrix}1 & 2\\ 1 & -1\end{bmatrix}$. Also, we suppose the filter is parameterized by a weight array $\mathbf{W}$ with the same size of $\mathbf{a}$, i.e., $|\mathbf{a}_p| = |\mathbf{W}|$. The result of applying the filter to $\mathbf{A}$ is an array of features

$$\text{Conv}(\mathbf{A},\mathbf{W}) = \begin{bmatrix}v_1 & \dots & v_{|\Omega|}\end{bmatrix} \tag{4.37}$$

[9] In mathematical analysis, given two integrable functions $f(\cdot)$ and $g(\cdot)$, **convolution** defines a new integrable function $f*g(\cdot)$ to describe the integral of $f(\cdot)$ weighted by reflected, shifted $g(\cdot)$. More formally, the convolution for continuous functions is defined as

$$f*g(x) = \int_{\mathbb{R}} f(y)g(x-y)\mathrm{d}y \tag{4.35}$$

where $f(y)$ is the function that we are concerned with, and $g(x-y)$ is the weight function which is translated by reflecting $g(y)$ along the y-axis and then shifting it by x. A special case is that x and y are both integers. In this case, we can define $f*g(\cdot)$ as

$$f*g(x) = \sum_y f(y)g(x-y) \tag{4.36}$$

which is the basic form of Eq. (4.33). In CNNs, x, y and $x-y$ can be seen as indices of items in data arrays. $f(y)$ is a data item in the input array, and $g(x-y)$ is the corresponding weight in the filter. By using Eq. (4.36), we calculate the value of the item indexed by x in the output array $f*g(x)$ (i.e., the feature map).

Each feature v_p is given by

$$\begin{aligned} v_p &= \text{Conv}(\mathbf{a}_p, \mathbf{W}) \\ &= \mathbf{a}_p \cdot \mathbf{W} \\ &= \sum_{k=1}^{|\mathbf{W}|} a_p(k) \cdot W(k) \end{aligned} \tag{4.38}$$

where $a_p(k)$ and $W(k)$ are the k-th elements of $\mathbf{a}_p$ and $\mathbf{W}$, respectively. Note that the array $\begin{bmatrix} v_1 & ... & v_{|\Omega|} \end{bmatrix}$ can be organized into different shapes, such as a matrix or a 3D tensor, though they are essentially the same thing from the data storage viewpoint. For example, for 2D input data and a 2D filter, the feature map is a matrix like Eq. (4.34).

Furthermore, we need to consider two things to make the model practical. First, we need to specify the stride of each move of the filter over $\mathbf{A}$. In the above example, we simply use $stride = 1$. By choosing a larger stride, we can compress $\mathbf{A}$ into a smaller number of features. Second, in some situations, to ensure that the feature map has a desired size, we can add dummy elements (or paddings) around the input data. A common method of padding is to set zeros to the elements outside the input region. For example, consider a 2×2 data matrix.

$$\mathbf{A} = \begin{bmatrix} 1 & 9 \\ 7 & 3 \end{bmatrix} \tag{4.39}$$

We can add zero-valued entries around it to obtain a 4×4 matrix, like this

$$\mathbf{A}_{\text{padding}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 9 & 0 \\ 0 & 7 & 3 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.40}$$

Using the same filter as in Eq. (4.33) with $stride = 1$, we have a 3×3 feature map

$$\text{Conv}_{stride=1}(\mathbf{A}_{\text{padding}}, \mathbf{W}) = \begin{bmatrix} 2 & 20 & 18 \\ 14 & 22 & 24 \\ 0 & 14 & 6 \end{bmatrix} \tag{4.41}$$

If $stride = 2$, then we would have a feature map with the same size of the input data

$$\text{Conv}_{stride=2}(\mathbf{A}_{\text{padding}}, \mathbf{W}) = \begin{bmatrix} 2 & 18 \\ 0 & 6 \end{bmatrix} \tag{4.42}$$

4.4.2 CNNs for Sequence Modeling

Following the formulation in the previous sections, we assume that the input of a sequence model is a vector sequence $\mathbf{x}_1 ... \mathbf{x}_m$ and the output is another vector sequence $\mathbf{h}_1 ... \mathbf{h}_m$. For example, we can think of $\mathbf{x}_1 ... \mathbf{x}_m$ as a matrix $\mathbf{X} \in \mathbb{R}^{m \times d_e}$ in which the i-th row vector is the

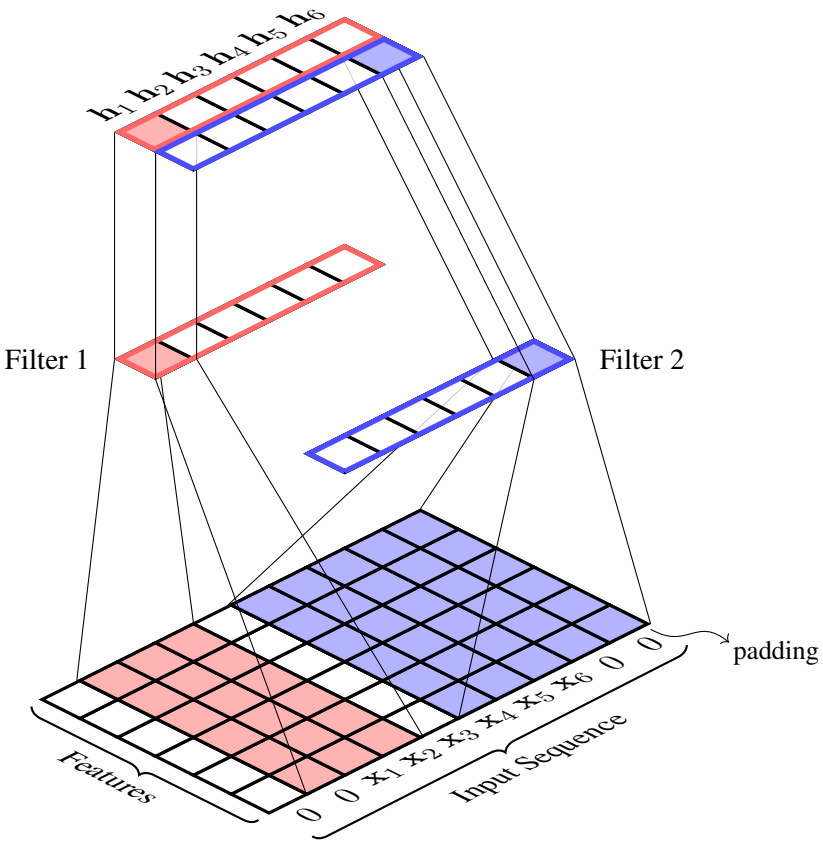


Figure 4.7: Two filters applied to a sequence of word vectors. The input involves ten word vectors (words $\mathbf{x}_1 ... \mathbf{x}_6$ and two padding words on each of the two ends of the sequence). Each word vector has 6 dimensions, and so, the input is a 10×6 matrix. Filter 1 has a receptive field of size 3×6. By sliding it over the input matrix, we obtain a sequence of outputs, each corresponding to a position (i.e., a sequence of 6 outputs). Similarly, we apply filter 2 to the input sequence and obtain another sequence of outputs. The two output sequences are then organized as a 2×6 matrix in which the i-th row vector is $\mathbf{h}_i$.

representation of the i-th word of the sequence.

It is straightforward to perform convolution on $\mathbf{X}$. Since $\mathbf{x}_i$ is just a set of unordered features, it is not necessary to slide a filter over different features. Hence we can use a receptive field of size $r \times d_e$, and consider all the dimensions of $\mathbf{x}_i$ in the convolution operation. In practical applications, there might be multiple filters for representing the inputs in different aspects. For example, one can use a filter with a large receptive field to involve more contexts in modeling, and use a filter with a small receptive field to concentrate more on local features. See Figure 4.7 for two filters that are used to deal with a sequence.

To distribute features to $\{\mathbf{h}_1, ..., \mathbf{h}_m\}$, we can associate each application of a filter to a position of the sequence. To ensure the input and output sequences are of the same length, a padding vector is added to each end of the sequence. The following shows the input and output

of a CNN for an example sequence.

Position	Input	Receptive Field	Output
0	$\mathbf{x}_0(=0)$	N/A	N/A
1	$\mathbf{x}_1$	$\{\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2\}$	$\mathbf{h}_1$
2	$\mathbf{x}_2$	$\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$	$\mathbf{h}_2$
3	$\mathbf{x}_3$	$\{\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$	$\mathbf{h}_3$
4	$\mathbf{x}_4$	$\{\mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5\}$	$\mathbf{h}_4$
5	$\mathbf{x}_5(=0)$	N/A	N/A

An activation function is typically used to introduce some non-linearity to the final output. In this way, we build a standard convolutional layer which can be viewed as a sequence of fully connected neural networks, each taking inputs from a fixed-size window. For the i-th position, the output of the convolutional layer is given by

$$v_i \quad = \quad \psi(\text{Conv}(\mathbf{a}_i, \mathbf{W})) \tag{4.43}$$

where $\mathbf{a}_i$ is the inputs in the receptive field[10], and $\mathbf{W}$ is the parameters of the filter. In situations involving multiple filters (say d_h filters), we have a set of parameters $\{\mathbf{W}^{(1)}, ..., \mathbf{W}^{(d_h)}\}$, a set of activation functions $\{\psi^{(1)}, ..., \psi^{(d_h)}\}$, and a set of inputs $\{\mathbf{a}_i^{(1)}, ..., \mathbf{a}_i^{(d_h)}\}$. Each tuple $(\mathbf{W}^{(k)}, \psi^{(k)}, \mathbf{a}_i^{(k)})$ gives an output by

$$v_i^{(k)} \quad = \quad \psi^{(k)}(\text{Conv}(\mathbf{a}_i^{(k)}, \mathbf{W}^{(k)})) \tag{4.44}$$

Note that $v_i^{(k)}$ is simply an entry of $\mathbf{h}_i$. Thus, $\mathbf{h}_i$ can be written as

$$\mathbf{h}_i \quad = \quad \begin{bmatrix} v_i^{(1)} & ... & v_i^{(d_h)} \end{bmatrix} \tag{4.45}$$

Many CNN-based systems of practical interest comprise two or more convolutional layers. The simplest way to achieve this is layer stacking, as in multi-layer RNNs (see Section 4.2.3). That is, we treat the output of a convolutional layer as the input of the following layer. See Figure 4.8 for an example of a CNN involving three convolutional layers. One of the benefits of multi-layer CNNs is a larger scope for representation. As seen from Figure 4.8, a neuron in layer 1 connects three input vectors, while a neuron in layer 3 connects, though not directly, seven input vectors. Since neurons of the higher-level layers receive and process signals from a larger span of the sequence, they are expected to produce a higher-level representation of the sequence and to be able to deal with more difficult problems, such as long-distance dependencies.

In some applications, we need a fixed-length, low-dimensional representation of the entire sequence. A common way is to add a pooling layer to merge $\{\mathbf{h}_1, ..., \mathbf{h}_m\}$ into a single vector. For example, we can select the maximum value (i.e., max-pooling) or average the values (i.e.,

[10] For an $r \times d_e$ receptive field, $\mathbf{a}_i$ is defined to be $\{\mathbf{x}_{\lceil i-\frac{r}{2} \rceil}, ..., \mathbf{x}_{\lceil i+\frac{r}{2}-1 \rceil}\}$ or $\{\mathbf{x}_{\lfloor i-\frac{r}{2} \rfloor}, ..., \mathbf{x}_{\lfloor i+\frac{r}{2}-1 \rfloor}\}$.

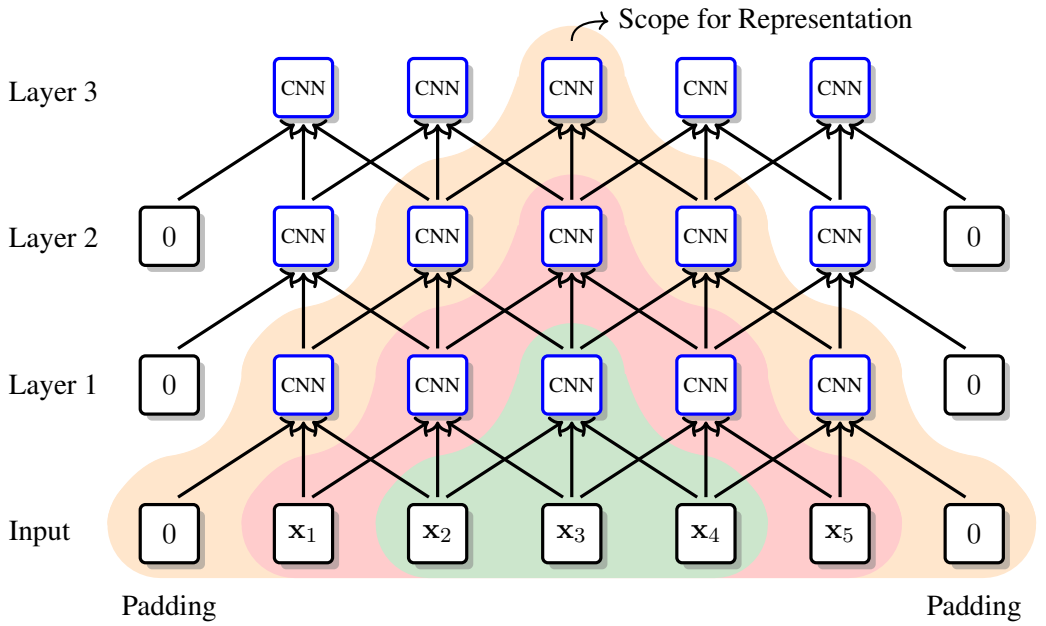


Figure 4.8: A CNN with 3 convolutional layers ($stride = 1$ and $r = 3$). Each layer takes a sequence of vectors and produces another sequence of vectors. In this process a filter moves over the input and performs the convolution operation in each move. In layer 1, the receptive field of the filter is a region of three input items (see green shadows). The higher a layer is, the larger receptive field a filter has. For example, in layer 3, an application of the filter can at most cover the entire input sequence (see orange shadows).

max-pooling) along each dimension. This is a generic method in machine learning and is applicable to most of the sequence models discussed in this book.

4.4.3 Handling Positional Information

One interesting property of CNNs is their ability to balance complexity and efficiency. This is achieved by restricting full connectivity to only a small region of the input data. This also leads us to describe a convolution layer using the same mathematical form of a layer in a fully-connected neural network: the output of a neuron is some transformation of the weighted sum of the input numbers. Despite the simplicity inherent in modeling, a problem with such models is that the order of inputs is completely ignored. An interesting point, however, is that, if we restrict ourselves to sequence modeling, this should not be a problem because the output of the model is itself a sequence. It seems reasonable to assume that the output sequence preserves the ordering information of the input sequence. On the other hand, applying CNNs to sequential data does not guarantee a one-to-one mapping between the input and output items. Technically, $\mathbf{h}_i$ is not simply a representation of $\mathbf{x}_i$. It instead encodes a window of inputs centered at $\mathbf{x}_i$. This, in turn, makes the problem very complicated, since it is difficult to work out from $\mathbf{h}_i$ how those inputs are ordered.

Explicitly modeling word orders is very important in NLP, and has been extensively studied in tasks like machine translation [Lopez, 2008; Koehn, 2010]. For neural network-based models, one may address the problem by resorting to order-sensitive model architectures like RNNs. A more popular approach in recent systems is to develop a **positional encoding** sub-model and incorporate it into existing sequence models [Gehring et al., 2017b; Vaswani et al., 2017; Shaw et al., 2018; Dufter et al., 2022]. Formally, we say that the input at position i is a combination of the original input $\mathbf{x}_i$ and the positional encoding of i (denoted by $\text{PE}(\cdot)$):

$$\mathbf{xp}_i \quad = \quad \text{Merge}(\mathbf{x}_i, \text{PE}(i)) \tag{4.46}$$

where $\text{Merge}(\cdot)$ combines $\mathbf{x}_i$ and $\text{PE}(i)$ in some way. The use of positional encoding is straightforward: all you need is to replace $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$ with $\{\mathbf{xp}_1, ..., \mathbf{xp}_m\}$ in a sequence model. So, this approach is model-free.

In this subsection, we present several versions of $\text{PE}(\cdot)$ and ways to combine them with $\mathbf{x}_i$. Note that the following discussion is not specific to CNNs. We consider it here because positional encoding is useful for models that are insensitive to the order of inputs, and CNNs are a good example to see how it is used [Gehring et al., 2017a;b]. In Chapter 6, we will see an application of positional encoding in Transformer which is a state-of-the-art neural model in many areas.

1. Offset-based Positional Encoding

The simplest way to describe a position i is to just leave it as it is. This can be formulated as the "distance" from a reference point

$$\text{PE}(i) \quad = \quad i - i_0 \tag{4.47}$$

where i_0 is an integer indicating where we start counting. If $i_0 = 0$, $\text{PE}(i) = i$ gives the normal way to define a position. Note that $\text{PE}(i)$ could be a negative number if $i_0 > i$. In this sense, $\text{PE}(i)$ is not a real distance but it is fine with considering it as a feature in a machine learning system. To design a non-negative measure, the right-hand side of Eq. (4.47) can be defined as an absolute value

$$\text{PE}(i) \quad = \quad |i - i_0| \tag{4.48}$$

Treating positions as simple integers leads to unbounded, discrete positional encoding. A more desirable method might be to use a continuous representation in a range of values, because it allows the system to work within a sample space that is smooth and easy to optimize. A simple way to do this is normalization. For example, dividing $i - i_0$ by some maximum value, we obtain a normalized version of the offset-based encoding

$$\text{PE}(i) \quad = \quad \frac{i - i_0}{i_{\max} - i_0} \tag{4.49}$$

For example, we can set $i_{\max}$ = the maximum possible length of the sequence and $i_0 = 0$,

so that $\text{PE}(i)$ chooses its value in $[0, 1]$. Another common choice is to set $i_{\max} = m$ (i.e., the length of the input sequence) and define $\text{PE}(i)$ as a ratio whose value varies as m changes.

To make use of these scalar positions, it is straightforward to enrich the original input vectors by adding new dimensions, provided they can be viewed as new features. Thus, $\mathbf{xp}_i$ is given by

$$\mathbf{xp}_i = [\mathbf{x}_i, \text{PE}(i)] \tag{4.50}$$

where $[\cdot]$ stands for the concatenation operation.

2. Sinusoidal Positional Encoding

The next obvious step is to represent positions as vectors instead of scalars. Although vectorizing the representations of positions sounds complicated, a simple idea is to use a carrying system which describes how a natural number is expressed by a polynomial with respect to a base [Kernes, 2021]. For example, i can be written as

$$i = \sum_{k=0}^{k_{\max}} a(i,k) b^k \tag{4.51}$$

where $a(i,k)$ is the k-th digit, $k_{\max}+1$ is the maximum number of digits, and b is the base of the system. The carrying occurs when $a(i,k)$ reaches b: we increase $a(i,k+1)$ by 1 and roll back $a(i,k)$ to 0. In this way we can change $a(i,k)$ with a period of b^k, that is, $a(i,0)$ changes with a period of b^0, $a(i,1)$ changes with a period of b^1, $a(i,2)$ changes with a period of b^2, and so on.

Using this system, i can be represented as a vector

$$\text{PE}(i) = \begin{bmatrix} a(i,0) & a(i,1) & \dots & a(i,k_{\max}) \end{bmatrix} \tag{4.52}$$

For example, when $b = 2$, $\text{PE}(11) = \begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix}$. However, in Eq. (4.52), $\text{PE}(i)$ is still a discrete function. As discussed throughout this book, we may want a continuous vector representation that can describe intermediate states between discrete events. Considering $a(i,k)$ as a periodic function, a common choice is the **sine** function. Thus $a(i,k)$ can be re-defined, as follows

$$a(i,k) = \sin(i \cdot \omega_k) \tag{4.53}$$

This function has an amplitude of 1 and a period of $\frac{2\pi}{\omega_k}$. Using an analogous form of periods to that used in Eq. (4.51), we define ω_k as

$$\omega_k = \frac{1}{(b_{\text{model}})^{k/d_{\text{model}}}} \tag{4.54}$$

where $b_{\text{model}} > 0$ and $d_{\text{model}} > 0$ are hyper-parameters of the model. Obviously, we have $\frac{2\pi}{\omega_0} < \frac{2\pi}{\omega_1} < \dots < \frac{2\pi}{\omega_{k_{\max}}}$.

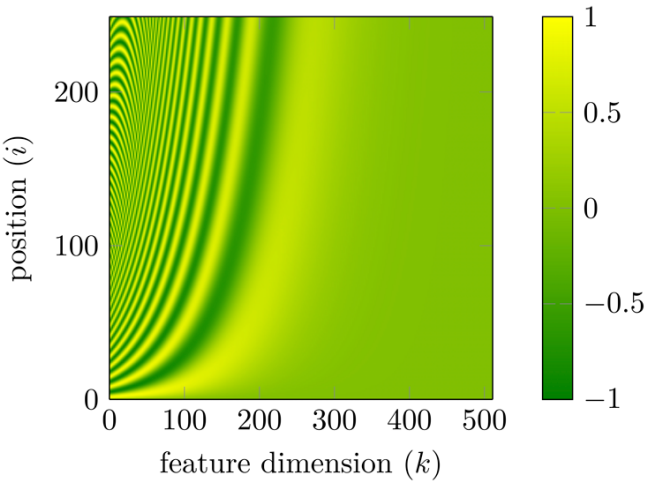


Figure 4.9: A heat map of the positional embedding model of Eqs. (4.57) and (4.58) ($b_{\text{model}} = 10,000$ and $d_{\text{model}} = 512$). Consider a position i (i.e., the i-th row), then move another position j from i upwards or downwards. Intuitively, when i and j are closer, the corresponding row vectors are more similar. By contrast, when j moves away from i, the similarity is not that obvious. This property helps explain the idea behind the positional embedding model: the "distance" between two positions is implicitly modeled by comparing their multi-dimensional representations.

Similarly, we can define $a(i,k)$ via the **cosine** function

$$a(i,k) \quad = \quad \cos(i \cdot \omega_k) \tag{4.55}$$

Taking both Eqs. (4.53) and (4.55), we create a new representation of i, as follows

$$\text{PE}(i) \quad = \quad \Big[\sin(i \cdot \omega_0) \quad \cos(i \cdot \omega_0) \quad ... \quad \sin(i \cdot \omega_{k_{\max}}) \quad \cos(i \cdot \omega_{k_{\max}})\Big] \tag{4.56}$$

Vaswani et al. [2017] instantiated the above form by setting $b_{\text{model}} = 10,000$. Let $\text{PE}(i,k)$ be the k-th dimension of $\text{PE}(i)$. Vaswani et al. [2017]'s version of positional encoding is written as

$$\text{PE}(i,2k) \quad = \quad \sin(i \cdot \frac{1}{10000^{2k/d_{\text{model}}}}) \tag{4.57}$$

$$\text{PE}(i,2k+1) \quad = \quad \cos(i \cdot \frac{1}{10000^{2k/d_{\text{model}}}}) \tag{4.58}$$

Choosing $b_{\text{model}} = 10,000$ is empirical. One can adjust it for specific tasks. Figure 4.9 plots the positional encoding for different positions. We see that, when k becomes larger, the change of the color follows a larger period.

Note that Eqs. (4.57) and (4.58) have a useful property that $\text{PE}(i+\mu)$ can be easily

expressed by a linear function of $\text{PE}(i)$ for a given offset μ[11]

$$\begin{aligned} \text{PE}(i+\mu, 2k) &= \text{PE}(i, 2k) \cdot \text{PE}(\mu, 2k+1) + \\ &\quad \text{PE}(i, 2k+1) \cdot \text{PE}(\mu, 2k) \end{aligned} \tag{4.61}$$

$$\begin{aligned} \text{PE}(i+\mu, 2k+1) &= \text{PE}(i, 2k+1) \cdot \text{PE}(\mu, 2k+1) + \\ &\quad \text{PE}(i, 2k) \cdot \text{PE}(\mu, 2k) \end{aligned} \tag{4.62}$$

The resulting benefit is that the encoding can somewhat model relative positions. That is, the state at position $i+\mu$ can be described by starting with i and then appending it with the offset μ.

When applying the sinusoidal positional encoding, one way is to use Eq. (4.50) to concatenate $\mathbf{x}_i$ and $\text{PE}(i)$. In Vaswani et al. [2017]'s work, they instead assume $\text{PE}(i)$ to be a vector of the same size as $\mathbf{x}_i$ (i.e., $|\text{PE}(i)| = |\mathbf{x}_i| = d_e$), and add $\text{PE}(i)$ to $\mathbf{x}_i$, like this

$$\mathbf{xp}_i = \mathbf{x}_i + \text{PE}(i) \tag{4.63}$$

This sinusoidal additive model has been the basis of many positional encoding approaches [Dehghani et al., 2018; Likhomanenko et al., 2021; Su et al., 2021].

3. Learnable Positional Encoding

The result of sinusoidal positional encoding is a lookup table $\mathbf{C}_{\text{PE}} \in \mathbb{R}^{m_{\max} \times d_e}$ (where m_{max} is the maximum sequence length we can choose)

$$\mathbf{C}_{\text{PE}} = \begin{bmatrix} \text{PE}(1) \\ \ldots \\ \text{PE}(m_{\max}) \end{bmatrix} \tag{4.64}$$

In this table, each row vector $\text{PE}(i)$ corresponds to the embedding of a position i. These vectors, as described above, are computed based on some assumptions and heuristic algorithms. An alternative approach is to treat vectors of positions as parameters of the model and learn them as usual. In this case, both word embeddings and position embeddings can be trained in the same manner. See Chapters 2 and 3 for more information about learning word embeddings in neural language models.

One last note on positional encoding. What we have shown in this section can broadly be characterized as an **absolute positional encoding** paradigm: a position is described by its location in a coordinate system. Another concept that is worth exploring is **relative positional encoding** [Shaw et al., 2018]. For example, we can extend Eq. (4.48) to define the distance

[11] One can derive these by taking

$$\sin(\alpha+\beta) = \sin(\alpha) \cdot \cos(\beta) + \cos(\alpha) \cdot \sin(\beta) \tag{4.59}$$

$$\cos(\alpha+\beta) = \cos(\alpha) \cdot \cos(\beta) - \sin(\alpha) \cdot \sin(\beta) \tag{4.60}$$

between two positions i and j

$$\text{PE}(i,j) \quad = \quad |i-j| \tag{4.65}$$

In this case, the positional encoding is no longer an attribute of the i-th input but some representation of the distance relative to a reference position j. In fact, most of the methods for relative positional encoding are variants on a theme in which positions are described by their pair-wise relationships. This forms the basis of several models of this type as we will see in Chapter 6.

4.5 Examples

Both recurrent and convolutional models have been successfully used in numerous applications. Here we discuss a few of the interesting examples. While these models are mostly basic, they form the foundations of many state-of-the-art systems.

4.5.1 Text Classification

To illustrate how sequence models could be used, we first consider the text classification problem in which we assign one of some pre-defined classes to a text. It can be extended to cover a broad range of problems in NLP, including classifying news texts, flagging sentiment sentences, identifying spam emails, detecting fake comments, and so on.

In text classification we are interested in selecting the best class from a set C, given a word sequence $w_1...w_m$:

$$\hat{c} \quad = \quad \underset{c \in C}{\arg\max}\ \text{Score}(c, w_1...w_m) \tag{4.66}$$

Here $\text{Score}(c, w_1...w_m)$ measures how well a class c is predicted for the input sequence $w_1...w_m$. Here we map the sequence of words to the sequence of word vectors (or word embeddings), that is, $w_1...w_m \rightarrow \mathbf{x}_1...\mathbf{x}_m$. Assuming $\text{Score}(\cdot)$ is a probabilistic function that describes the distribution of the classes, we can reformulate the problem as

$$\begin{aligned} \hat{c} \quad &= \quad \underset{c \in C}{\arg\max}\ \Pr(c|w_1...w_m) \\ &= \quad \underset{c \in C}{\arg\max}\ \Pr(c|\mathbf{x}_1...\mathbf{x}_m) \end{aligned} \tag{4.67}$$

The central issue here is the modeling of $\Pr(c|\mathbf{x}_1...\mathbf{x}_m)$. We define $\Pr(c|\mathbf{x}_1...\mathbf{x}_m)$ by following the general encoder + predictor framework, as follows

- The input $\mathbf{x}_1...\mathbf{x}_m$ is represented as a feature vector $\mathbf{H} \in \mathbb{R}^{d_h}$ by using an encoder

$$\mathbf{H} \quad = \quad \text{Encoder}(\mathbf{x}_1...\mathbf{x}_m) \tag{4.68}$$

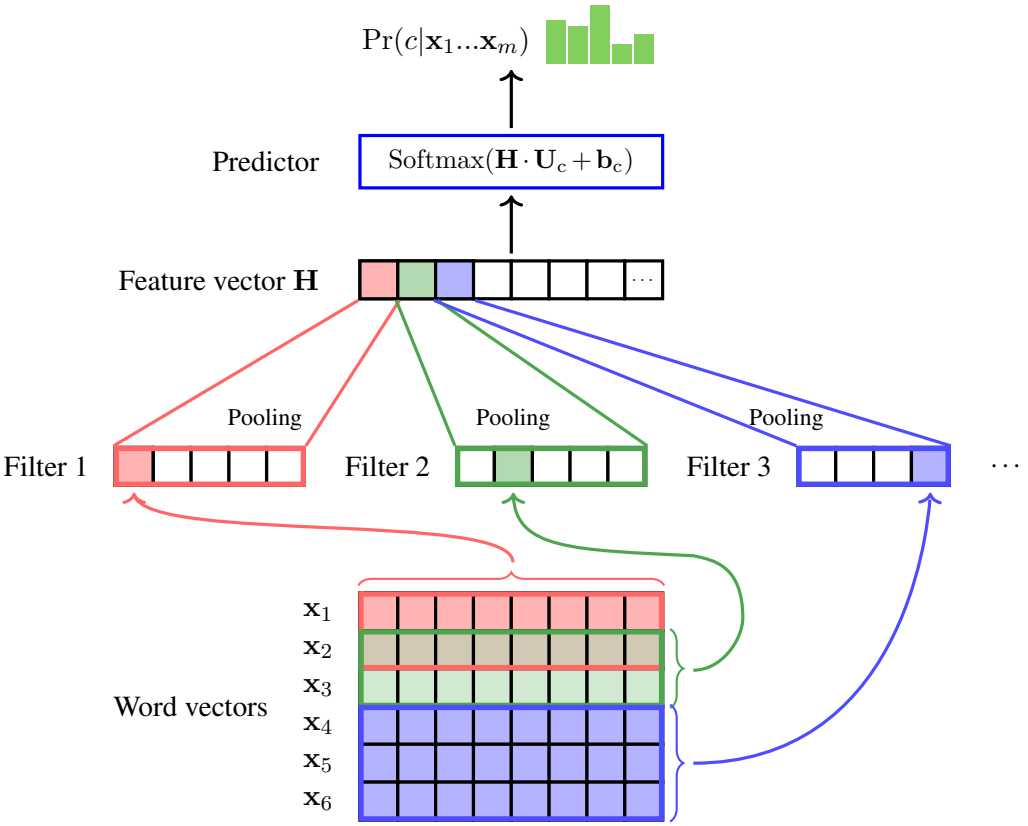


Figure 4.10: A CNN-based text classifier [Kim, 2014]. The input is a sequence of word vectors. A convolutional layer involving multiple filters is used to extract features in different dimensions. A pooling layer is used to reduce the number of features for representing the input text, leading to a low-dimensional feature vector $\mathbf{H}$. The prediction conditions on $\mathbf{H}$ and is made by using a standard Softmax layer.

- $\mathbf{H}$ is fed to a standard Softmax layer to predict the class distribution

$$\Pr(\cdot|\mathbf{H}) \quad = \quad \text{Softmax}(\mathbf{H} \cdot \mathbf{U}_c + \mathbf{b}_c) \tag{4.69}$$

  where $\mathbf{U}_c \in \mathbb{R}^{d_h \times |C|}$ and $\mathbf{b}_c \in \mathbb{R}^{|C|}$ are model parameters.

$\text{Encoder}(\cdot)$ is exactly the same thing we discussed in the preceding sections. There are, therefore, many encoding models that are applicable here. For example, consider the CNN-based encoder presented in Kim [2014]'s work. Kim [2014]'s model is based on a single convolution layer involving d_h filters. The application of a filter produces a set of features, each being associated with a position of the sequence. Since we want a single vector for representing the entire sequence, a pooling layer is added so that the number of features for each filter is reduced to one. Then, for any c, the probability $\Pr(c|\mathbf{H})$ can be computed trivially according to Eq. (4.69). See Figure 4.10 for an illustration of this classifier.

To train such a model, we just need to optimize it on some loss, and, as mentioned several

times in this book, one of the most common methods is to minimize the cross-entropy loss using gradient descent. Also, we can use regularization to improve the training of CNNs. More details about these techniques can be found in Chapter 2.

Note that while the model described here is quite "simple", it is among the most effective models known for text classification. There are, of course, improvements to this kind of classifier. Examples of such systems include deep CNNs [Conneau et al., 2017b], character-based CNNs [Santos and Gatti, 2014; Zhang et al., 2015], recurrent CNNs [Lai et al., 2015], and so on.

4.5.2 End-to-End Speech Recognition

Speech recognition is a task of taking a sequence of acoustic signals and mapping it to a sequence of words or characters (call it a **transcription**) [Reddy, 1976; Rabiner and Juang, 1993]. Since the original input is an acoustic waveform over the time domain, it is common to transform it into a sequence of waveform fragments (call them **frames**). Typically, a frame is represented as a feature vector, denoted by $\mathbf{x}_i$. This is achieved by using either feature functions in signal processing [Davis and Mermelstein, 1980; Picone, 1993; Campbell, 1997] or learnable embeddings [Chorowski et al., 2019; Schneider et al., 2019]. Regarding the output, speech recognition systems generally do not produce words. Instead, they produce sequences of **transcription units** (or **transcription labels**), e.g., phonemes, characters, sub-words, etc. In this subsection we assume that the output of a speech recognition system is a sequence of English letters, denoted by $y_1...y_n \in V_{\mathrm{y}}^n$. The alphabet V_{y} consists of normal English letters ($\mathrm{a}-\mathrm{z}$), numbers ($0-9$), spaces ($\langle \mathrm{sp} \rangle$), periods ($\langle \mathrm{pe} \rangle$), and other punctuation marks. As with most modern speech recognition systems, we add a blank symbol ϵ to the alphabet in order to indicate the null output.

The goal here is to find a string $\hat{y}_1...\hat{y}_n$ for a given input sequence $\mathbf{x}_1...\mathbf{x}_m$, so that

$$\hat{y}_1...\hat{y}_n \quad = \quad \arg\max_{y_1...y_n} \Pr(y_1...y_n|\mathbf{x}_1...\mathbf{x}_m) \tag{4.70}$$

This model is relatively difficult compared to the classification model described in the previous subsection, as the output can be an arbitrary string, rather than a class in a predefined class set. The string generation problem leads to two difficulties. First, we need some mechanism to model $\Pr(y_1...y_n|\mathbf{x}_1...\mathbf{x}_m)$ for an exponentially large number of pairs of input and output sequences. Second, in practice $y_1...y_n$ is often much shorter than $\mathbf{x}_1...\mathbf{x}_m$ (i.e., $n < m$), and so we need some mechanism to align a long sequence to a short one. However, we do not need to consider these difficulties in the stage of representing the input sequence, and can still encode the input sequence $\mathbf{x}_1...\mathbf{x}_m$ in the same way as other sequence models. Specifically, we represent $\mathbf{x}_1...\mathbf{x}_m$ in the following form

$$\mathbf{h}_1...\mathbf{h}_m \quad = \quad \mathrm{Encoder}(\mathbf{x}_1...\mathbf{x}_m) \tag{4.71}$$

The encoder can be RNNs, CNNs, or more advanced models (such as Transformer). Here we consider the encoder architecture used in Graves et al. [2013b] and Graves and Jaitly

[2014]'s work. It is a multi-layer bi-directional LSTM model. We skip the details of this model without loss of continuity, as the reader may be already familiar with it in Sections 4.2.3, 4.2.4 and 4.3.2.

Having obtained the sequence representation $\mathbf{H} = \mathbf{h}_1 ... \mathbf{h}_m$, a softmax layer is used to map each $\mathbf{h}_i \in \mathbb{R}^{d_h}$ to a distribution of transcription labels, given by

$$\Pr(\cdot|\mathbf{h}_i) \quad = \quad \text{Softmax}(\mathbf{h}_i \cdot \mathbf{U}_\text{s} + \mathbf{b}_\text{s})$$

where $\mathbf{U}_\text{s} \in \mathbb{R}^{d_h \times |V_\text{y}|}$ and $\mathbf{b}_\text{s} \in \mathbb{R}^{|V_\text{y}|}$ are model parameters. $\Pr(\cdot|\mathbf{h}_i) \in \mathbb{R}^{|V_\text{y}|}$ is a probability distribution over V_y, and the probability of transcription label l_i at position i is simply $\Pr(l_i|\mathbf{h}_i)$. We can then write the probability of a label sequence in the form

$$\Pr(l_1 ... l_m|\mathbf{H}) \quad = \quad \prod_{k=1}^{m} \Pr(l_k|\mathbf{h}_i) \tag{4.72}$$

This formulation looks simple. We can appeal to the $\arg\max$ operation to find the most probable label sequence as usual. However, $l_1 ... l_m$ cannot be straightforwardly used as a system output, because it often contains many duplicate and blank symbols. To "post-process" $l_1 ... l_m$, we first merge the sub-sequence of labels to a single label when they are the same, and then remove the blank symbols. For example, consider a label sequence

$$\text{s} \quad \text{s} \quad \epsilon \quad \text{e} \quad \text{e} \quad \epsilon \quad \epsilon \quad \text{e} \quad \langle\text{sp}\rangle \quad \text{y} \quad \epsilon \quad \text{o} \quad \text{o} \quad \text{u}$$

By merging "s s", "e e", and "o o", we have

$$\text{s} \quad \epsilon \quad \text{e} \quad \epsilon \quad \epsilon \quad \text{e} \quad \langle\text{sp}\rangle \quad \text{y} \quad \epsilon \quad \text{o} \quad \text{u}$$

Then, we remove all ϵ and obtain

$$\text{s} \quad \text{e} \quad \text{e} \quad \langle\text{sp}\rangle \quad \text{y} \quad \text{o} \quad \text{u}$$

The above sequence is what we would call a transcription. Obviously, different label sequences can correspond to the same transcription. Let $B(y_1 ... y_n)$ be the set of label sequences corresponding to $y_1 ... y_n$[12]. We now turn to the following form of the transcription probability (see Figure 4.11 for an illustration)

$$\Pr(y_1 ... y_n|\mathbf{x}_1 ... \mathbf{x}_m) \quad = \quad \sum_{l_1 ... l_m \in B(y_1 ... y_n)} \Pr(l_1 ... l_m|\mathbf{H}) \tag{4.73}$$

A problem with this model is that the number of the sequences in $B(y_1 ... y_n)$ grows exponentially with n (and m). Fortunately, there are very efficient methods for computing $\sum_{l_1 ... l_m \in B(y_1 ... y_n)} \Pr(l_1 ... l_m|\mathbf{H})$. See [Graves et al., 2006] for a dynamic programming

[12] $B(y_1 ... y_n)$ may contain label sequences of arbitrary lengths. However, if we restrict input to the sequence $\mathbf{x}_1 ... \mathbf{x}_m$, then each sequence in $B(y_1 ... y_n)$ is of length m.

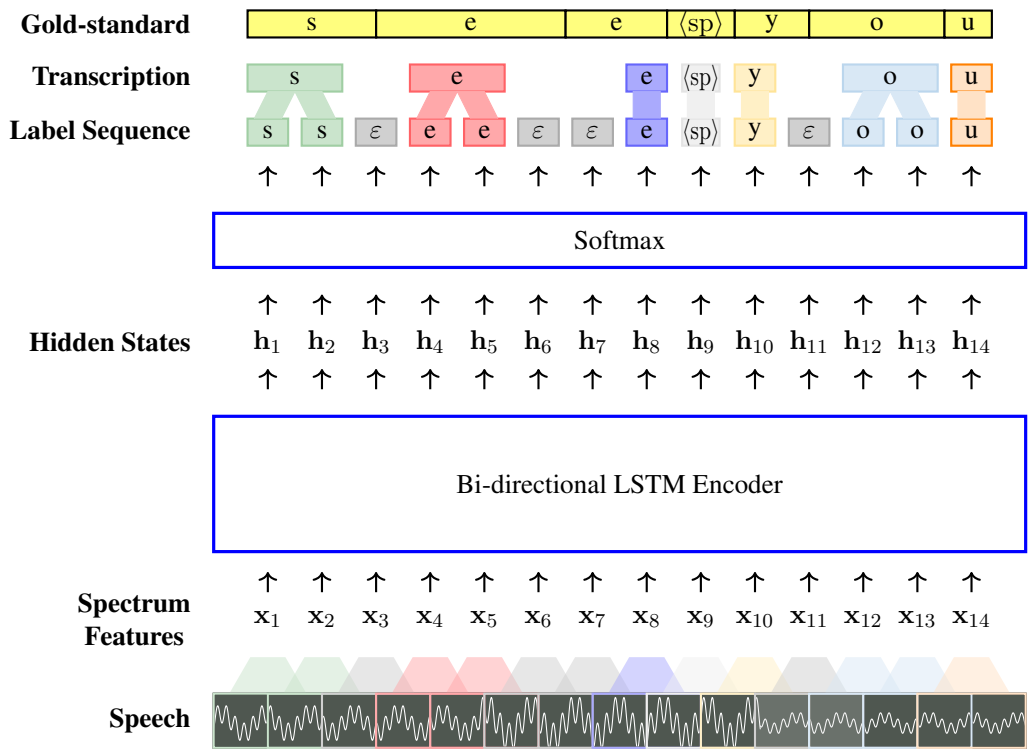


Figure 4.11: An end-to-end speech recognition architecture. The input of the system is a sequence of acoustic signals that are represented as a sequence of feature vectors (i.e.., $\mathbf{x}_1 ... \mathbf{x}_{14}$). These feature vectors are taken by a bi-directional LSTM encoder. The output of the encoder is a sequence of contextualized representations (i.e., $\mathbf{h}_1 ... \mathbf{h}_{14}$) which is then fed into a softmax layer for generating a distribution of labels at each position. We can then draw a sequence of labels from these distributions. We map each label sequence to a form of final output by eliminating duplicate symbols and blank symbols. An output of the system corresponds to a number of label sequences, and the probability of the output is the sum of the probabilities of these label sequences.

algorithm for solving this problem.

Note that Eq. (4.73) is also known as a form of **connectionist temporal classification** (**CTC**) [Graves et al., 2006]. It is one of the most widely used methods for training end-to-end speech recognition and speech translation systems. One of the merits of CTC is that it allows us to align any label sequence to a transcription in a very simple and efficient way. It is easy to make use of CTC in training a speech recognition system. Suppose there is a set of pairs of input sequence and transcription, denoted by S. A common training objective is to maximize the likelihood of these transcriptions given the corresponding inputs, written as

$$\hat{\theta} = \arg\max_{\theta} \sum_{(y_1 ... y_n, \mathbf{x}_1 ... \mathbf{x}_m) \in S} \log \Pr(y_1 ... y_n | \mathbf{x}_1 ... \mathbf{x}_m; \theta) \tag{4.74}$$

where $\Pr(y_1 ... y_n | \mathbf{x}_1 ... \mathbf{x}_m; \theta)$ is the probability computed via Eq. (4.73), and θ is the parameters

of the model.

When testing on new data, we search for an optimal transcription as in Eq. (4.70). This process, also known as **decoding**, generally involves optimized search algorithms and pruning techniques. For example, we can use the 1-best label sequence instead of all possible label sequences to obtain an approximation to Eq. (4.73), that is, $\Pr(y_1...y_n|\mathbf{x}_1...\mathbf{x}_m) = \max \Pr(l_1...l_m|\mathbf{H})$. This leads to an efficient decoding method, called **Viterbi decoding**, which has been extensively used in speech recognition and machine translation [Lopez, 2008]. For more details about the decoding of sequence generation, we refer the reader to Chapter 5.

4.5.3 Sequence Labeling with LSTM and Graphical Models

Sequence labeling is a conceptually straightforward approach to classifying data in sequence. In NLP, it has penetrated many sub-areas like word segmentation, part-of-speech tagging, and chunking. Learning in these models consists of simply predicting a label in a label set V_{y} at each position of a sequence. Ideally, we wish to perform a sequence of labeling actions based on the entire input, given by

$$\hat{y}_1...\hat{y}_m \quad = \quad \underset{y_1...y_m}{\arg\max} \Pr(y_1...y_m|\mathbf{x}_1...\mathbf{x}_m) \tag{4.75}$$

where $\mathbf{x}_1...\mathbf{x}_m$ is an input sequence (such as a sequence of word vectors), and $y_1...y_m$ is a label sequence in which each label y_i corresponds to an input item $\mathbf{x}_i$.

As we have seen in this chapter, Eq. (4.75) perfectly fits the form of the sequence modeling problem. As a first step we use an encoder to map $\mathbf{x}_1...\mathbf{x}_m$ to a sequence of contextualized representations, as follows

$$\mathbf{h}_1...\mathbf{h}_m \quad = \quad \mathrm{Encoder}(\mathbf{x}_1...\mathbf{x}_m) \tag{4.76}$$

We define $\mathrm{Encoder}(\cdot)$ as a bi-directional LSTM model because it involves a memory mechanism for modeling long-range dependencies in both left and right contexts. Hence, the architecture of the encoder is the same as that used in the preceding subsection.

$\mathbf{h}_1...\mathbf{h}_m$ can then be taken to be the input of a usual sequence labeling system (see Figure 4.12). The sequence labeling problem has been discussed in Chapter 1, and many models are applicable to it. The simplest is the one that involves a classifier (such as maximum entropy and SVM-based models) for predicting a label distribution for each $\mathbf{h}_i$. A problem with these models is that the predictions are made independently. A more powerful approach is to use **graphical models** to consider dependencies among predicted labels. For example, **hidden Markov models** (**HMMs**) describe how a sequence of observations (i.e., $\mathbf{x}_1...\mathbf{x}_m$) is generated given a sequence of variables (i.e., $y_1...y_m$). The key idea is to rewrite $\Pr(y_1...y_m|\mathbf{x}_1...\mathbf{x}_m)$ using the Bayes' rule and approximate $\Pr(y_1...y_m|\mathbf{x}_1...\mathbf{x}_m)$ by a product of simple factors. However, these models require probability density functions of continuous variables (e.g., $\Pr(\mathbf{x}_i|y_i)$) which are difficult to estimate. This differentiates the use of HMMs in neural models greatly from that in conventional models where all states and observations are discrete

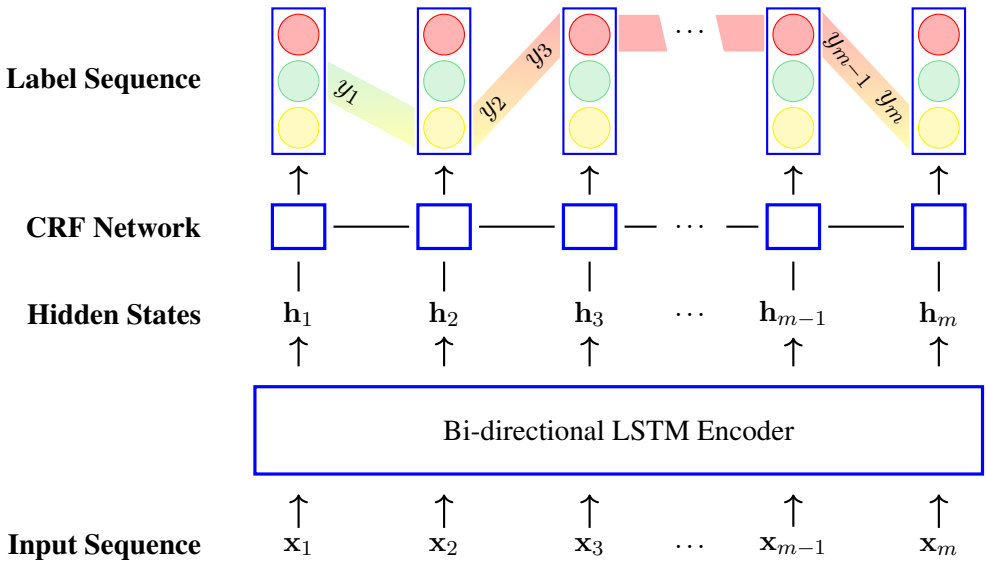


Figure 4.12: The BiLSTM + graphical model architecture for sequence labeling. The encoder is a standard bi-directional LSTM model. Given a sequence of input feature vectors (i.e., $\mathbf{x}_1 ... \mathbf{x}_m$), it produces a new sequence of feature vectors for mapping the input to contextualized representations (i.e., $\mathbf{h}_1 ... \mathbf{h}_m$). A CRF network is placed on $\mathbf{h}_1 ... \mathbf{h}_m$ to predict a distribution of label sequences. The optimal label sequence is the one that has the maximum probability.

variables[13].

HMMs and their descendants can be viewed as instances of generative models. Another type of model that has been commonly used to solve sequence labeling problems is discriminative models. One such model is **conditional random fields** (**CRFs**) [Lafferty et al., 2001]. The CRF model features its ability to directly model the joint probability of the entire input and

[13]In HMMs, a sequence of variables can be viewed as a path of transiting over some states whose values are chosen from a pre-defined set. In each transition from one state to another, something is observed (call it an observation). By making some assumptions, we can approximate $\Pr(y_1 ... y_m | \mathbf{x}_1 ... \mathbf{x}_m)$ in the following fashion

$$\begin{aligned} \Pr(y_1 ... y_m | \mathbf{x}_1 ... \mathbf{x}_m) &= \frac{\Pr(y_1 ... y_m) \cdot \Pr(\mathbf{x}_1 ... \mathbf{x}_m | y_1 ... y_m)}{\Pr(\mathbf{x}_1 ... \mathbf{x}_m)} \\ &\approx \frac{\prod_{i=1}^{m} \Pr(y_i | y_{i-1}) \cdot \prod_{i=1}^{m} \Pr(\mathbf{x}_i | y_i)}{\Pr(\mathbf{x}_1 ... \mathbf{x}_m)} \\ &= \frac{\prod_{i=1}^{m} \Pr(y_i | y_{i-1}) \Pr(\mathbf{x}_i | y_i)}{\Pr(\mathbf{x}_1 ... \mathbf{x}_m)} \end{aligned} \tag{4.77}$$

where $\Pr(y_i | y_{i-1})$ is the **transition probability** of moving from y_{i-1} to y_i (when $i = 1$, we define $\Pr(y_i | y_{i-1}) = \Pr(y_1 | y_0) = \Pr(y_1)$), and $\Pr(\mathbf{x}_i | y_i)$ is the **emission probability** of observing $\mathbf{x}_i$ given y_i. As the denominator $\Pr(\mathbf{x}_1 ... \mathbf{x}_m)$ is a constant number for different $y_1 ... y_m$, it can be dropped in the argmax operation of Eq. (4.75), as follows

$$\hat{y}_1 ... \hat{y}_m = \arg\max_{y_1 ... y_m} \prod_{i=1}^{m} \Pr(y_i | y_{i-1}) \Pr(\mathbf{x}_i | y_i) \tag{4.78}$$

To estimate $\Pr(\mathbf{x}_i | y_i)$, a possible solution is to take $\Pr(\mathbf{x}_i | y_i) = \frac{\Pr(y_i | \mathbf{x}_i) \Pr(\mathbf{x}_i)}{\Pr(y_i)}$, and use a neural network to compute $\Pr(y_i | \mathbf{x}_i)$.

label sequences, and to allow us to make use of a variety of features to do this. Consider, for example, the **linear-chain CRF** [Sutton and McCallum, 2012]. It defines $\Pr(y_1...y_m|\mathbf{h}_1...\mathbf{h}_m)$ in the following form

$$\begin{aligned}\Pr(y_1...y_m|\mathbf{h}_1...\mathbf{h}_m) &= \frac{\Pr(y_1...y_m, \mathbf{h}_1...\mathbf{h}_m)}{\Pr(\mathbf{h}_1...\mathbf{h}_m)} \\ &= \frac{\exp(\text{Score}(y_1...y_m, \mathbf{h}_1...\mathbf{h}_m))}{Z(\mathbf{h}_1...\mathbf{h}_m)}\end{aligned} \tag{4.79}$$

where $Z(\mathbf{h}_1...\mathbf{h}_m)$ is a normalization factor, and has the form

$$Z(\mathbf{h}_1...\mathbf{h}_m) = \sum_{y_1^{'}...y_m^{'}} \exp(\text{Score}(y_1^{'}...y_m^{'}, \mathbf{h}_1...\mathbf{h}_m)) \tag{4.80}$$

$\text{Score}(\cdot)$ is a score for weighting the sequence pair $(y_1...y_m, \mathbf{h}_1...\mathbf{h}_m)$. It is given by summing over the values of a set of feature functions $\{f_1(\cdot), ..., f_J(\cdot)\}$, like this

$$\text{Score}(y_1...y_m, \mathbf{h}_1...\mathbf{h}_m) = \sum_{i=1}^{m}\sum_{j=1}^{J} f_j(y_i, y_{i-1}, \mathbf{h}_i) \tag{4.81}$$

The outer loop of the summation corresponds to a visit to each position i. Given i, each function $f_j(\cdot)$ takes the current label y_i, the previous label y_{i-1} and the current input vector $\mathbf{h}_i$, and then returns the value of a feature.

This model is called *linear-chain* because it represents $y_1...y_m$ as a chain structure where each node y_i, along with an observed variable $\mathbf{h}_i$, only connects to its preceding node y_{i-1} and its following node y_{i+1}[14], like this

$$\begin{array}{ccccccccccccc} y_1 & — & \cdots & — & y_{i-1} & — & y_i & — & y_{i+1} & — & \cdots & — & y_m \\ | & & & & | & & | & & | & & & & | \\ \mathbf{h}_1 & & \cdots & & \mathbf{h}_{i-1} & & \mathbf{h}_i & & \mathbf{h}_{i+1} & & \cdots & & \mathbf{h}_m \end{array}$$

In CRFs, it is assumed that the variables in the graph is only dependent on its neighboring variables. Therefore, $f_j(y_i, y_{i-1}, \mathbf{h}_i)$ can be defined according to how y_i is connected. There are generally two types of features.

- **Transition-like Features**. This type of features models the connection between consecutive labels (y_{i-1}, y_i), given by

$$f_1(y_i, y_{i-1}, \mathbf{h}_i) = u(y_{i-1}, y_i) \tag{4.82}$$

  where $u(y_{i-1}, y_i)$ is an entry of a weight matrix $\mathbf{u}$, indexed by (y_{i-1}, y_i).

- **Emission-like Features**. The second type of features models the connection between a

[14] CRFs can broadly be categorized as a type of **undirected graphical models**. They define a graph over a set of observed variables and a set of unobserved variables. These variables are connected in some way that forms a graph.

label y_i and the associated input $\mathbf{x}_i$, given by

$$f_2(y_i, y_{i-1}, \mathbf{h}_i) \quad = \quad g_i(y_i) \tag{4.83}$$

where $g_i(y_i)$ is the entry y_i of a vector $\mathbf{g}_i \in \mathbb{R}^{|V_y|}$. The vector $\mathbf{g}_i$ represents the weights of associating $\mathbf{h}_i$ with each label in the form

$$\mathbf{g}_i \quad = \quad \mathbf{h}_i \cdot \mathbf{v} \tag{4.84}$$

where $\mathbf{v} \in \mathbb{R}^{d_h \times |V_y|}$ is a weight matrix.

To simplify notation, we use y_i (or y_{i-1}) to denote the one-hot representation for a label[15]. Then, substituting the above feature functions into Eq. (4.81) allows the scoring function to be written in the form

$$\begin{aligned} \text{Score}(y_1 ... y_m, \mathbf{h}_1 ... \mathbf{h}_m) &= \sum_{i=1}^{m} u(y_{i-1}, y_i) + g_i(y_i) \\ &= \sum_{i=1}^{m} y_{i-1} \cdot \mathbf{u} \cdot y_i^{\mathrm{T}} + \mathbf{h}_i \cdot \mathbf{v} \cdot y_i^{\mathrm{T}} \\ &= \sum_{i=1}^{m} (y_{i-1} \cdot \mathbf{u} + \mathbf{h}_i \cdot \mathbf{v}) \cdot y_i^{\mathrm{T}} \end{aligned} \tag{4.85}$$

The right-hand side of the equation only involves simple algebraic operations on vectors and matrices, allowing viewing this model as a normal neural network. In this way, it is convenient to implement the sequence labeling system with various neural network toolkits. We just need to stack a CRF network on an encoder network and learn the entire network as usual. For example, one can train this system by maximum likelihood, and optimize the loss function by gradient descent. Note that, as with other chain or lattice-based models, the CRF network can be efficient because there are dynamic programming algorithms, called the forward-backward methods, for computing both $\text{Score}(y_1 ... y_m, \mathbf{h}_1 ... \mathbf{h}_m)$ and $Z(\mathbf{h}_1 ... \mathbf{h}_m)$. We refer the interested reader to related papers for more detailed discussions [Lafferty et al., 2001; Sutton and McCallum, 2012].

One advantage of marrying the worlds of distributed representation and sequence labeling is that we do not need to specify any feature templates as in conventional approaches. Instead, the model is free to learn features that describe whatever input sequences are most effective at optimizing some objective for sequence labeling. Such an architecture has been used as the backbone model for several state-of-the-art systems [Huang et al., 2015; Lample et al., 2016; Ma and Hovy, 2016; Li et al., 2020].

[15] In this case, $y_i \in \mathbb{R}^{|V_y|}$, although it is originally used as a scalar.

4.5.4 Hybrid Models for Language Modeling

As we have already noted, many sequence modeling problems can be dealt with by either RNN-based or CNN-based models. Each of these two types of models has its own characteristics: RNNs are originally designed for dealing with variable-length temporal data, and CNNs are more effective in interpreting local information in restricted regions of input. Here we consider a hybrid approach to language modeling for obtaining the benefits of both.

Recall from Section 4.2.1 that a neural language model is learned to predict a probability distribution over a vocabulary, given some representation of the history words. Let $w_1...w_m$ be a word sequence to which we want to assign a probability. First, we represent each word w_i as a word vector $\mathbf{x}_i$. Then, an RNN model takes a word vector at a time and outputs the probability

$$\begin{aligned} \Pr(w_{i+1}|w_1...w_i) &= \Pr(w_{i+1}|\mathbf{x}_1...\mathbf{x}_i) \\ &= \Pr(w_{i+1}|\mathbf{h}_i) \end{aligned} \tag{4.86}$$

where $\mathbf{h}_i$ is the state of the recurrent unit at step i.

The process of converting words from symbols to continuous representations plays an important role in this model. While it is common for practitioners to obtain $\mathbf{x}_i$ from a word embedding table, this approach treats each word as a whole and simply ignores its internal structure. In consequence, it might be difficult to learn distinct vectors for rare words in languages with large vocabularies [Bojanowski et al., 2017].

Here we consider a different way of representing words. The idea is simple: an additional neural network is used to embed words [Ling et al., 2015; Kim et al., 2016]. Suppose every word w_i can be expressed as a sequence of characters. We represent these characters as real-valued vectors $\mathbf{e}_{i,1}...\mathbf{e}_{i,\text{len}_i}$ via a character embedding table. Following Kim et al. [2016]'s work, we can use a CNN to represent $\mathbf{e}_{i,1}...\mathbf{e}_{i,\text{len}_i}$ as a word vector in the following form

$$\begin{aligned} \mathbf{x}_i &= \text{CNN}(\mathbf{e}_{i,1}...\mathbf{e}_{i,\text{len}_i}, \mathbf{W}) \\ &= \text{Pooling}(\text{TanH}(\text{Conv}(\mathbf{e}_{i,1}...\mathbf{e}_{i,\text{len}_i}, \mathbf{W}))) \end{aligned} \tag{4.87}$$

where $\text{Conv}(\cdot, \mathbf{W})$ is a convolutional layer with parameters $\mathbf{W}$, $\text{TanH}(\cdot)$ is a hyperbolic tangent function, and $\text{Pooling}(\cdot)$ is a pooling layer.

Figure 4.13 shows the architecture of the model. We see that there is a hierarchical structure behind this model, that is, characters form a word, and words form a sentence or phrase. On a practical side, in many NLP tasks it is quite natural to consider the hierarchical nature of language. We will see a few examples of making use of the relationships between different levels of language representations in later chapters.

4.6 Summary

This chapter has introduced the recurrent and convolutional neural approaches to modeling sequences of words. On one hand, recurrent neural networks are designed for dealing with

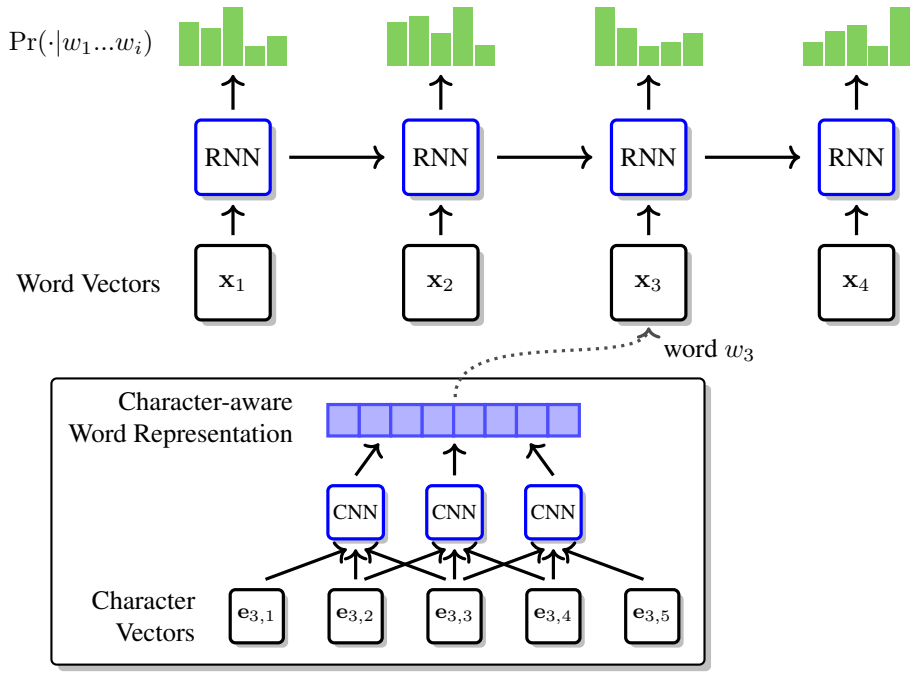


Figure 4.13: A language model with character-aware word representations [Kim et al., 2016]. As a language model, the goal of this model is to compute the probability $\Pr(w_{i+1}|w_1...w_i)$ for each i. We represent each word w_i as a real-valued vector $\mathbf{x}_i$. This vector is the output of a CNN that takes a sequence of characters corresponding to this word. Then, the sequence of the word vectors $\mathbf{x}_1...\mathbf{x}_m$ is used as the input to an RNN + Softmax model. The model outputs at each position i a distribution of words, where the entry w_{i+1} describes $\Pr(w_{i+1}|w_1...w_i)$. This hierarchical structure provides a multi-scale approach to language modeling: a sentence is modeled by considering words, and a word is modeled by considering characters.

sequential data, and have broad applicability in NLP. To improve the modeling power of these models, the memory mechanism is generally used. In particular, we have introduced LSTM and GRU which are two popular types of models in dealing with long sequence problems. On the other hand, while convolutional neural networks are commonly used to process vision data, they are straightforwardly applicable to sequence modeling. We have seen that all these models can be used in several language and speech processing tasks, including text classification, speech recognition, sequence labeling, and language modeling.

The roots of modeling sequences of language units can be traced back to early work in several different fields. For example, the process of generating a sequence of words can be described as a Markov chain where the prediction of a word only depends on a limited number of previous words [Markov, 1913]. This idea motivates the n-gram methods for sequence modeling [Shannon, 1948], as well as hidden Markov models which later appeared and became popular in modeling sequences of pairs of observed and unobserved variables [Baum and Petrie, 1966; Baum et al., 1970]. These models and their variants lay the foundations of many successful NLP systems in past decades [Manning and Schütze, 1999; Jurafsky and Martin,

2008].

The idea of using neural networks in sequence modeling has also been investigated for some time. One example to see how neural networks are developed and applied to sequence modeling is speech recognition [Lippmann, 1989]. Most of the studies in the early days of this research area try to either combine neural networks with existing models [Bourlard and Wellekens, 1990; Bourlard and Morgan, 1993; Trentin and Gori, 2001], or address sub-problems of speech recognition [Tank and Hopfield, 1987; Waibel et al., 1989; Lang et al., 1990; Bengio, 1991]. However, scaling neural networks up in size was challenging because training deep neural networks requires a lot of computation resources and data. The field had long been dominated by approaches based on hidden Markov models and **Gaussian mixture models** (**GMMs**), with a pipeline of several modules that require careful tuning. On the other hand, while fully neural approaches were not state-of-the-art during that time, researchers were aware of their potential in learning representations of acoustic inputs and freeing them from hand-crafted features [LeCun and Bengio, 1995].

A dramatic shift from conventional pipelined approaches to end-to-end approaches comes with the revival of neural networks in the 2000s [Hannun et al., 2014; Graves et al., 2013b]. The shift is so influential that a broad set of fields comes together like never before, e.g., in computer vision and speech processing, the past ten years have, meanwhile, witnessed great performance gains brought by very deep neural networks and end-to-end learning [Hinton et al., 2006; Graves et al., 2013b; He et al., 2016; Krizhevsky et al., 2017]. In NLP, the paradigm shift starts with the work on word embeddings [Mikolov et al., 2013; Pennington et al., 2014], and continues as more powerful sequence representation models are developed [Vaswani et al., 2017]. A simple approach to sequence modeling, though not discussed in depth in this chapter, is **compositional models** [Janssen, 2012]. For example, we can use the bag-of-words model to sum or average word vectors of a sequence. Despite the simple architectures of these approaches, they achieve satisfactory results in many tasks, providing strong baselines for further research on more advanced models [Conneau et al., 2018]. As the next step, applying recurrent and convolutional neural models to sequence modeling is straightforward. This is not surprising because these models are fairly well studied in other fields [Lipton et al., 2015; Li et al., 2021b; Khan et al., 2020]. In particular, the LSTM model is well suited to deal with long sequences and thus of great interest to NLP researchers [Sundermeyer et al., 2012; Huang et al., 2015; Wu et al., 2016]. However, we are always on the way. Learning sequence models is one of the most active research fields with no end in sight. There are many models that are based on new architectures and show stronger performance in various tasks. More discussions on some of these models can be found in Chapters 6, 7 and 8.

Note that the term *sequence modeling* is currently used in many different ways, referring to different tasks. In many cases it is more common to use the terms *encoding* and *encoder* to emphasize the process of mapping a sequence of symbols to a continuous representation. As discussed in the previous sections, a benefit of viewing encoding as an individual task is that we can learn a general representation model that is not dependent on where we apply it. It opens the door to a wide range of pre-trained encoders for learning to represent various types of data, such as text [Peters et al., 2018; Devlin et al., 2019], speech [Oord et al., 2018;

Hsu et al., 2021; Chen et al., 2022], vision [Chen and He, 2021; Bao et al., 2021; He et al., 2022], and combinations of them [Chuang et al., 2020; Li et al., 2021a; Kim et al., 2021]. A closely related concept to text encoding is **text embedding** or **sentence embedding** [Conneau et al., 2017a; Cer et al., 2018]. These can be broadly considered the same thing. In general, an embedding model in NLP means a process of transforming the input text into a single low-dimensional vector rather than producing sequences of contextualized vectors [Kiros et al., 2015; Hill et al., 2016].

In many NLP problems, systems are not necessarily sequential on their input and/or output. For example, in text classification, a system may take tree-structured input and produces a label [Tai et al., 2015; Yang et al., 2016]. In this case we need some mechanism to encode hierarchical structures. An alternative approach is to convert trees to sequences (or **linearized trees**) so that we can directly make use of sequence models to handle non-sequential data [Vinyals et al., 2015]. This is a great idea because it opens up the possibility of developing a universally applicable encoder to represent various types of data if the input of the encoder can be linearized in some way. For example, by representing an image as a sequence of patches, sequence models can be directly applied to image classification, achieving state-of-the-art results on several tasks [Chen et al., 2020; Dosovitskiy et al., 2021].

Bibliography

[Atkinson and Shiffrin, 1968] Richard C Atkinson and Richard M Shiffrin. Human memory: A proposed system and its control processes. In *Psychology of learning and motivation*, volume 2, pages 89–195. Elsevier, 1968.

[Bao et al., 2021] Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. Beit: Bert pre-training of image transformers. In *Proceedings of International Conference on Learning Representations*, 2021.

[Baum and Petrie, 1966] Leonard E Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state markov chains. *The annals of mathematical statistics*, 37(6):1554–1563, 1966.

[Baum et al., 1970] Leonard E Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The annals of mathematical statistics*, 41(1):164–171, 1970.

[Bengio, 1991] Yoshua Bengio. *Artificial neural networks and their application to sequence recognition*. PhD thesis, McGill University, 1991.

[Bengio et al., 1994] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[Bishop, 2006] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Bojanowski et al., 2017] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the association for computational linguistics*, 5:135–146, 2017.

[Bourlard and Wellekens, 1990] Herve Bourlard and Christian J Wellekens. Links between markov models and multilayer perceptrons. *IEEE Transactions on pattern analysis and machine intelligence*, 12(12):1167–1178, 1990.

[Bourlard and Morgan, 1993] Herve A. Bourlard and Nelson Morgan. *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer Academic Publishers, USA, 1993.

[Box et al., 2015] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control (4th ed.)*. John Wiley & Sons, 2015.

[Campbell, 1997] Joseph P Campbell. Speaker recognition: A tutorial. *Proceedings of the IEEE*, 85(9): 1437–1462, 1997.

[Cer et al., 2018] Daniel Cer, Yinfei Yang, Sheng yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.

[Chatfield, 2003] Chris Chatfield. *The analysis of time series: an introduction*. Chapman and hall/CRC,

2003.

[Chen et al., 2020] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. Generative pretraining from pixels. In *International conference on machine learning*, pages 1691–1703. PMLR, 2020.

[Chen et al., 2022] Sanyuan Chen, Chengyi Wang, Zhengyang Chen, Yu Wu, Shujie Liu, Zhuo Chen, Jinyu Li, Naoyuki Kanda, Takuya Yoshioka, Xiong Xiao, Long Zhou, Shuo Ren, Yanmin Qian, Yao Qian, Jian Wu, Michael Zeng, and Furu Wei. Wavlm: Large-scale self-supervised pre-training for full stack speech processing. *IEEE Journal of Selected Topics in Signal Processing*, 16(6):1505–1518, 2022.

[Chen and He, 2021] Xinlei Chen and Kaiming He. Exploring simple siamese representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15750–15758, 2021.

[Cho et al., 2014] Kyunghyun Cho, Bart van Merriënboer, Çağlar Guùlçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014.

[Chorowski et al., 2019] Jan Chorowski, Ron J Weiss, Samy Bengio, and Aäron Van Den Oord. Unsupervised speech representation learning using wavenet autoencoders. *IEEE/ACM transactions on audio, speech, and language processing*, 27(12):2041–2053, 2019.

[Chuang et al., 2020] Yung-Sung Chuang, Chi-Liang Liu, Hung-yi Lee, and Lin-shan Lee. Speechbert: An audio-and-text jointly learned language model for end-to-end spoken question answering. In *Proceedings of Interspeech 2020*, pages 4168–4172, 2020.

[Chung et al., 2014] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *Proceedings of NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

[Conneau et al., 2017] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 670–680, 2017a.

[Conneau et al., 2017] Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann Lecun. Very deep convolutional networks for text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 1107–1116, 2017b.

[Conneau et al., 2018] Alexis Conneau, Germán Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. What you can cram into a single vector: Probing sentence embeddings for linguistic properties. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2126–2136, 2018.

[Davis and Mermelstein, 1980] Steven Davis and Paul Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE transactions on acoustics, speech, and signal processing*, 28(4):357–366, 1980.

[Dehghani et al., 2018] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

[Devlin et al., 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[Dosovitskiy et al., 2021] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proceedings of ICLR 2021*, 2021.

[Dufter et al., 2022] Philipp Dufter, Martin Schmitt, and Hinrich Schütze. Position information in transformers: An overview. *Computational Linguistics*, 48(3):733–763, 2022.

[Elman, 1990] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[Fuller, 2009] Wayne A Fuller. *Introduction to statistical time series*. John Wiley & Sons, 2009.

[Gehring et al., 2017] Jonas Gehring, Michael Auli, David Grangier, and Yann Dauphin. A convolutional encoder model for neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 123–135, 2017a.

[Gehring et al., 2017] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In *International conference on machine learning*, pages 1243–1252. PMLR, 2017b.

[Gers et al., 2000] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[Goodfellow et al., 2016] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[Graves and Jaitly, 2014] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of International conference on machine learning*, pages 1764–1772, 2014.

[Graves et al., 2006] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.

[Graves et al., 2013] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with deep bidirectional lstm. In *IEEE workshop on automatic speech recognition and understanding*, pages 273–278. IEEE, 2013a.

[Graves et al., 2013] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013b.

[Graves et al., 2014] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[Graves et al., 2016] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adriá Puigdoménech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538

(7626):471–476, 2016.

[Hamilton, 1994] James Douglas Hamilton. *Time Series Analysis*. Princeton University Press, 1994.

[Hannun et al., 2014] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, and Adam Coates. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[He et al., 2016] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[He et al., 2022] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16000–16009, 2022.

[Hill et al., 2016] Felix Hill, Kyunghyun Cho, and Anna Korhonen. Learning distributed representations of sentences from unlabelled data. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1367–1377, 2016.

[Hinton et al., 2006] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[Hsu et al., 2021] Wei-Ning Hsu, Benjamin Bolte, Yao-Hung Hubert Tsai, Kushal Lakhotia, Ruslan Salakhutdinov, and Abdelrahman Mohamed. Hubert: Self-supervised speech representation learning by masked prediction of hidden units. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 29:3451–3460, 2021.

[Huang et al., 2015] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

[Ioffe and Szegedy, 2015] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[Jaeger and Haas, 2004] Herbert Jaeger and Harald Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *science*, 304(5667):78–80, 2004.

[Janssen, 2012] Theo M.V. Janssen. Compositionality: its historic context. In M. Werning, W. Hinzen, and E. Machery, editors, *The Oxford handbook of compositionality*. Oxford University Press, 2012.

[Jurafsky and Martin, 2008] Dan Jurafsky and James H. Martin. *Speech and Language Processing (2nd ed.)*. Prentice Hall, 2008.

[Kernes, 2021] Jonathan Kernes. Master positional encoding: Part i, 05 2021. URL https://towardsdatascience.com/master-positional-encoding-part-i-63c05d90a0c3.

[Khan et al., 2020] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial intelligence review*, 53 (8):5455–5516, 2020.

[Khandelwal et al., 2019] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike

Lewis. Generalization through memorization: Nearest neighbor language models. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2019.

[Kim et al., 2021] Wonjae Kim, Bokyung Son, and Ildoo Kim. Vilt: Vision-and-language transformer without convolution or region supervision. In *Proceedings of International Conference on Machine Learning*, pages 5583–5594. PMLR, 2021.

[Kim, 2014] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, October 2014.

[Kim et al., 2016] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. In *Proceedings of the Thirtieth AAAI conference on artificial intelligence*, 2016.

[Kiros et al., 2015] Ryan Kiros, Yukun Zhu, Russ R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. *Advances in neural information processing systems*, 28, 2015.

[Koehn, 2010] Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, 2010.

[Krizhevsky et al., 2017] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[Kumar et al., 2016] Ankit Kumar, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. In *International conference on machine learning*, pages 1378–1387, 2016.

[Lafferty et al., 2001] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning 2001*, pages 282–289, 2001.

[Lai et al., 2015] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. Recurrent convolutional neural networks for text classification. In *Twenty-ninth AAAI conference on artificial intelligence*, 2015.

[Lample et al., 2016] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 260–270, 2016.

[Lang et al., 1990] Kevin J Lang, Alex H Waibel, and Geoffrey E Hinton. A time-delay neural network architecture for isolated word recognition. *Neural networks*, 3(1):23–43, 1990.

[LeCun and Bengio, 1995] Yann LeCun and Yoshua Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[Li et al., 2020] Jing Li, Aixin Sun, Jianglei Han, and Chenliang Li. A survey on deep learning for named entity recognition. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):50–70, 2020.

[Li et al., 2021] Junnan Li, Ramprasaath Selvaraju, Akhilesh Gotmare, Shafiq Joty, Caiming Xiong, and Steven Chu Hong Hoi. Align before fuse: Vision and language representation learning with momentum distillation. *Advances in neural information processing systems*, 34:9694–9705, 2021a.

[Li et al., 2021] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of

convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 2021b.

[Likhomanenko et al., 2021] Tatiana Likhomanenko, Qiantong Xu, Gabriel Synnaeve, Ronan Collobert, and Alex Rogozhnikov. Cape: Encoding relative positions with continuous augmented positional embeddings. *Advances in Neural Information Processing Systems*, 34:16079–16092, 2021.

[Ling et al., 2015] Wang Ling, Chris Dyer, Alan W Black, Isabel Trancoso, Ramón Fermandez, Silvio Amir, Luis Marujo, and Tiago Luís. Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1520–1530, 2015.

[Lippmann, 1989] Richard P Lippmann. Review of neural networks for speech recognition. *Neural computation*, 1(1):1–38, 1989.

[Lipton et al., 2015] Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[Lopez, 2008] Adam Lopez. Statistical machine translation. *ACM Computing Surveys (CSUR)*, 40(3): 1–49, 2008.

[Ma and Hovy, 2016] Xuezhe Ma and Eduard Hovy. End-to-end sequence labeling via bi-directional lstm-cnns-crf. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1064–1074, 2016.

[Manning and Schütze, 1999] Chris Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[Manning et al., 2008] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[Markov, 1913] AA Markov. Essai d'une recherche statistique sur le texte du roman. *Eugene Onegin" illustrant la liaison des epreuve en chain ('Example of a statistical investigation of the text of "Eugene Onegin" illustrating the dependence between samples in chain"). In: Izvistia Imperatorskoi Akademii Nauk (Bulletin de l'Académie Impériale des Sciences de St.-Pétersbourg). 6th ser*, 7:153–162, 1913.

[Mikolov et al., 2010] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of Interspeech*, pages 1045–1048, 2010.

[Mikolov et al., 2013] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedings of the International Conference on Learning Representations (ICLR 2013)*, 2013.

[Miller et al., 2016] Alexander Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. Key-value memory networks for directly reading documents. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1400–1409, 2016.

[Mitchell, 1997] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Education, 1997.

[Oord et al., 2018] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

[Pascanu et al., 2013] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.

[Pennington et al., 2014] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[Peters et al., 2018] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 2018.

[Picone, 1993] Joseph W Picone. Signal modeling techniques in speech recognition. *Proceedings of the IEEE*, 81(9):1215–1247, 1993.

[Rabiner and Juang, 1993] Lawrence Rabiner and Biing-Hwang Juang. *Fundamentals of speech recognition*. Prentice-Hall, Inc., 1993.

[Reddy, 1976] D Raj Reddy. Speech recognition by machine: A review. *Proceedings of the IEEE*, 64 (4):501–531, 1976.

[Ross, 1924] William David Ross. *Aristotle's metaphysics*. Clarendon Press, 1924.

[Santos and Gatti, 2014] Cícero dos Santos and Maíra Gatti. Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 69–78, 2014.

[Schneider et al., 2019] Steffen Schneider, Alexei Baevski, Ronan Collobert, and Michael Auli. wav2vec: Unsupervised pre-training for speech recognition. In *INTERSPEECH*, 2019.

[Shannon, 1948] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[Shaw et al., 2018] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, 2018.

[Su et al., 2021] Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

[Sukhbaatar et al., 2015] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. *Advances in neural information processing systems*, 28, 2015.

[Sundermeyer et al., 2012] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Proceedings of the Thirteenth annual conference of the international speech communication association*, 2012.

[Sutskever, 2013] Ilya Sutskever. *Training recurrent neural networks*. University of Toronto Toronto, 2013.

[Sutton and McCallum, 2012] Charles Sutton and Andrew McCallum. An introduction to conditional random fields. *Foundations and Trends® in Machine Learning*, 4(4):267–373, 2012.

[Tai et al., 2015] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1556–1566, 2015.

[Tang et al., 2015] Duyu Tang, Bing Qin, and Ting Liu. Document modeling with gated recurrent

neural network for sentiment classification. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 1422–1432, 2015.

[Tank and Hopfield, 1987] David W Tank and JJ Hopfield. Neural computation by concentrating information in time. *Proceedings of the National Academy of Sciences*, 84(7):1896–1900, 1987.

[Trentin and Gori, 2001] Edmondo Trentin and Marco Gori. A survey of hybrid ann/hmm models for automatic speech recognition. *Neurocomputing*, 37(1-4):91–126, 2001.

[Vaswani et al., 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of Advances in Neural Information Processing Systems*, volume 30, 2017.

[Vinyals et al., 2015] Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. *Advances in neural information processing systems*, 28, 2015.

[Waibel et al., 1989] Alex Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing*, 37(3):328–339, 1989.

[Werbos, 1990] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[Weston et al., 2015] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. In *Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015*, 2015.

[Williams and Peng, 1990] Ronald J Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4):490–501, 1990.

[Wu et al., 2016] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[Yang et al., 2016] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 1480–1489, 2016.

[Zhang et al., 2021] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[Zhang et al., 2015] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. *Advances in neural information processing systems*, 28, 2015.