15

Modeling Sequential Data Using Recurrent Neural Networks

In the previous chapter, we focused on **convolutional neural networks** (**CNNs**). We covered the building blocks of CNN architectures and how to implement deep CNNs in PyTorch. Finally, you learned how to use CNNs for image classification. In this chapter, we will explore **recurrent neural networks** (**RNNs**) and see their application in modeling sequential data.

We will cover the following topics:

- Introducing sequential data
- RNNs for modeling sequences
- Long short-term memory
- Truncated backpropagation through time
- Implementing a multilayer RNN for sequence modeling in PyTorch
- Project one: RNN sentiment analysis of the IMDb movie review dataset
- Project two: RNN character-level language modeling with LSTM cells, using text data from Jules Verne's *The Mysterious Island*
- Using gradient clipping to avoid exploding gradients

Introducing sequential data

Let's begin our discussion of RNNs by looking at the nature of sequential data, which is more commonly known as sequence data or **sequences**. We will look at the unique properties of sequences that make them different from other kinds of data. We will then see how to represent sequential data and explore the various categories of models for sequential data, which are based on the input and output of a model. This will help us to explore the relationship between RNNs and sequences in this chapter.

Modeling sequential data — order matters

What makes sequences unique, compared to other types of data, is that elements in a sequence appear in a certain order and are not independent of each other. Typical machine learning algorithms for supervised learning assume that the input is **independent and identically distributed** (**IID**) data, which means that the training examples are *mutually independent* and have the same underlying distribution. In this regard, based on the mutual independence assumption, the order in which the training examples are given to the model is irrelevant. For example, if we have a sample consisting of n training examples, $\boldsymbol{x}^{(1)}$, $\boldsymbol{x}^{(2)}$, ..., $\boldsymbol{x}^{(n)}$, the order in which we use the data for training our machine learning algorithm does not matter. An example of this scenario would be the Iris dataset that we worked with previously. In the Iris dataset, each flower has been measured independently, and the measurements of one flower do not influence the measurements of another flower.

However, this assumption is not valid when we deal with sequences—by definition, order matters. Predicting the market value of a particular stock would be an example of this scenario. For instance, assume we have a sample of n training examples, where each training example represents the market value of a certain stock on a particular day. If our task is to predict the stock market value for the next three days, it would make sense to consider the previous stock prices in a date-sorted order to derive trends rather than utilize these training examples in a randomized order.

Sequential data versus time series data

Time series data is a special type of sequential data where each example is associated with a dimension for time. In time series data, samples are taken at successive timestamps, and therefore, the time dimension determines the order among the data points. For example, stock prices and voice or speech records are time series data.

On the other hand, not all sequential data has the time dimension. For example, in text data or DNA sequences, the examples are ordered, but text or DNA does not qualify as time series data. As you will see, in this chapter, we will focus on examples of natural language processing (NLP) and text modeling that are not time series data. However, note that RNNs can also be used for time series data, which is beyond the scope of this book.

Representing sequences

We've established that order among data points is important in sequential data, so we next need to find a way to leverage this ordering information in a machine learning model. Throughout this chapter, we will represent sequences as $\langle \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(T)} \rangle$. The superscript indices indicate the order of the instances, and the length of the sequence is T. For a sensible example of sequences, consider time series data, where each example point, $x^{(t)}$, belongs to a particular time, t. *Figure 15.1* shows an example of time series data where both the input features ($\boldsymbol{x}$'s) and the target labels ($\boldsymbol{y}$'s) naturally follow the order according to their time axis; therefore, both the $\boldsymbol{x}$'s and $\boldsymbol{y}$'s are sequences.

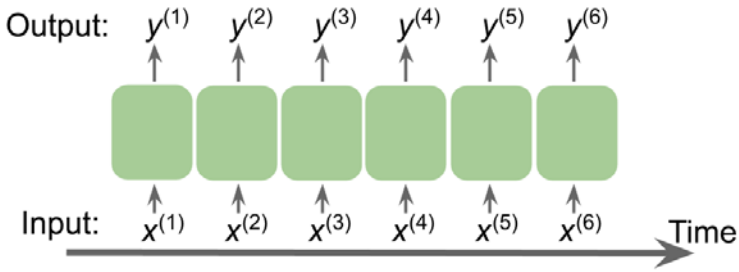

Figure 15.1: An example of time series data

As we have already mentioned, the standard NN models that we have covered so far, such as **multilayer perceptrons** (**MLPs**) and CNNs for image data, assume that the training examples are independent of each other and thus do not incorporate *ordering information*. We can say that such models do not have a *memory* of previously seen training examples. For instance, the samples are passed through the feedforward and backpropagation steps, and the weights are updated independently of the order in which the training examples are processed.

RNNs, by contrast, are designed for modeling sequences and are capable of remembering past information and processing new events accordingly, which is a clear advantage when working with sequence data.

The different categories of sequence modeling

Sequence modeling has many fascinating applications, such as language translation (for example, translating text from English to German), image captioning, and text generation. However, in order to choose an appropriate architecture and approach, we have to understand and be able to distinguish between these different sequence modeling tasks. *Figure 15.2*, based on the explanations in the excellent article *The Unreasonable Effectiveness of Recurrent Neural Networks*, by *Andrej Karpathy*, 2015 (`http://karpathy.github.io/2015/05/21/rnn-effectiveness/`), summarizes the most common sequence modeling tasks, which depend on the relationship categories of input and output data.

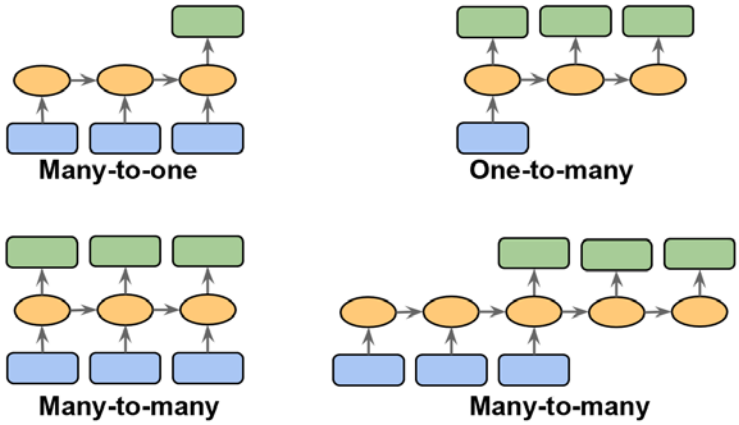

Figure 15.2: The most common sequencing tasks

Let's discuss the different relationship categories between input and output data, which were depicted in the previous figure, in more detail. If neither the input nor output data represent sequences, then we are dealing with standard data, and we could simply use a multilayer perceptron (or another classification model previously covered in this book) to model such data. However, if either the input or output is a sequence, the modeling task likely falls into one of these categories:

- **Many-to-one**: The input data is a sequence, but the output is a fixed-size vector or scalar, not a sequence. For example, in sentiment analysis, the input is text-based (for example, a movie review) and the output is a class label (for example, a label denoting whether a reviewer liked the movie).
- **One-to-many**: The input data is in standard format and not a sequence, but the output is a sequence. An example of this category is image captioning—the input is an image and the output is an English phrase summarizing the content of that image.
- **Many-to-many**: Both the input and output arrays are sequences. This category can be further divided based on whether the input and output are synchronized. An example of a synchronized many-to-many modeling task is video classification, where each frame in a video is labeled. An example of a *delayed* many-to-many modeling task would be translating one language into another. For instance, an entire English sentence must be read and processed by a machine before its translation into German is produced.

Now, after summarizing the three broad categories of sequence modeling, we can move forward to discussing the structure of an RNN.

RNNs for modeling sequences

In this section, before we start implementing RNNs in PyTorch, we will discuss the main concepts of RNNs. We will begin by looking at the typical structure of an RNN, which includes a recursive component to model sequence data. Then, we will examine how the neuron activations are computed in a typical RNN. This will create a context for us to discuss the common challenges in training RNNs, and we will then discuss solutions to these challenges, such as LSTM and **gated recurrent units** (**GRUs**).

Understanding the dataflow in RNNs

Let's start with the architecture of an RNN. *Figure 15.3* shows the dataflow in a standard feedforward NN and in an RNN side by side for comparison:

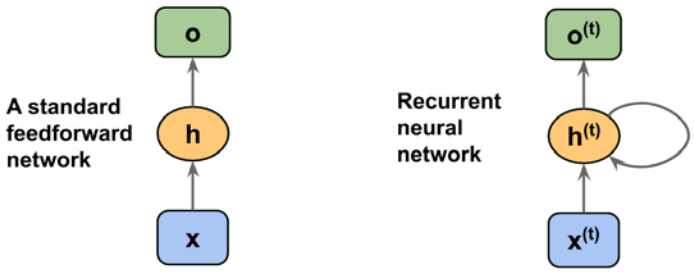

Figure 15.3: The dataflow of a standard feedforward NN and an RNN

Both of these networks have only one hidden layer. In this representation, the units are not displayed, but we assume that the input layer ($\boldsymbol{x}$), hidden layer ($\boldsymbol{h}$), and output layer ($\boldsymbol{o}$) are vectors that contain many units.

Determining the type of output from an RNN

This generic RNN architecture could correspond to the two sequence modeling categories where the input is a sequence. Typically, a recurrent layer can return a sequence as output, $\langle \boldsymbol{o}^{(0)}, \boldsymbol{o}^{(1)}, \ldots, \boldsymbol{o}^{(T)} \rangle$, or simply return the last output (at $t = T$, that is, $\boldsymbol{o}^{(T)}$). Thus, it could be either many-to-many, or it could be many-to-one if, for example, we only use the last element, $\boldsymbol{o}^{(T)}$, as the final output.

We will see later how this is handled in the PyTorch `torch.nn` module, when we take a detailed look at the behavior of a recurrent layer with respect to returning a sequence as output.

In a standard feedforward network, information flows from the input to the hidden layer, and then from the hidden layer to the output layer. On the other hand, in an RNN, the hidden layer receives its input from both the input layer of the current time step and the hidden layer from the previous time step.

The flow of information in adjacent time steps in the hidden layer allows the network to have a memory of past events. This flow of information is usually displayed as a loop, also known as a **recurrent edge** in graph notation, which is how this general RNN architecture got its name.

Similar to multilayer perceptrons, RNNs can consist of multiple hidden layers. Note that it's a common convention to refer to RNNs with one hidden layer as a *single-layer RNN*, which is not to be confused with single-layer NNs without a hidden layer, such as Adaline or logistic regression. *Figure 15.4* illustrates an RNN with one hidden layer (top) and an RNN with two hidden layers (bottom):

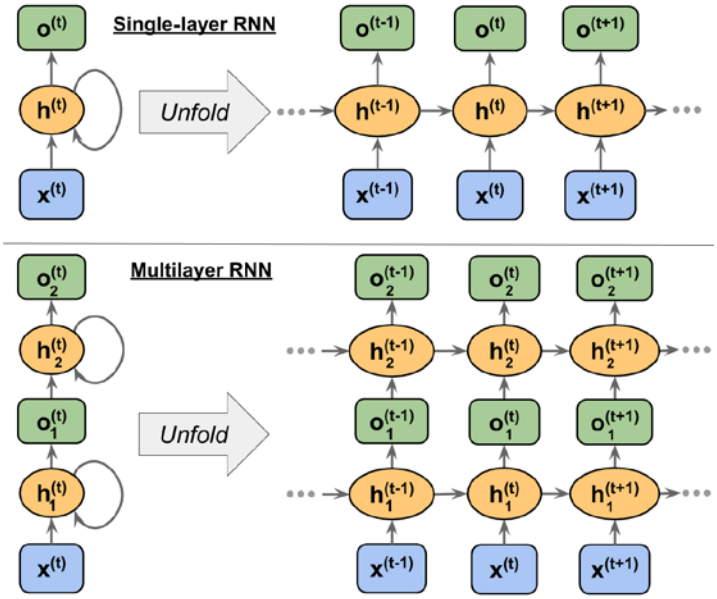

Figure 15.4: Examples of an RNN with one and two hidden layers

To examine the architecture of RNNs and the flow of information, a compact representation with a recurrent edge can be unfolded, which you can see in *Figure 15.4*.

As we know, each hidden unit in a standard NN receives only one input—the net preactivation associated with the input layer. In contrast, each hidden unit in an RNN receives two *distinct* sets of input—the preactivation from the input layer and the activation of the same hidden layer from the previous time step, $t - 1$.

At the first time step, $t = 0$, the hidden units are initialized to zeros or small random values. Then, at a time step where $t > 0$, the hidden units receive their input from the data point at the current time, $\boldsymbol{x}^{(t)}$, and the previous values of hidden units at $t - 1$, indicated as $\boldsymbol{h}^{(t-1)}$.

Similarly, in the case of a multilayer RNN, we can summarize the information flow as follows:

- *layer* = 1: Here, the hidden layer is represented as $\boldsymbol{h}_1^{(t)}$ and it receives its input from the data point, $\boldsymbol{x}^{(t)}$, and the hidden values in the same layer, but at the previous time step, $\boldsymbol{h}_1^{(t-1)}$.
- *layer* = 2: The second hidden layer, $\boldsymbol{h}_2^{(t)}$, receives its inputs from the outputs of the layer below at the current time step ($\boldsymbol{o}_1^{(t)}$) and its own hidden values from the previous time step, $\boldsymbol{h}_2^{(t-1)}$.

Since, in this case, each recurrent layer must receive a sequence as input, all the recurrent layers except the last one must *return a sequence as output* (that is, we will later have to set `return_sequences=True`). The behavior of the last recurrent layer depends on the type of problem.

Computing activations in an RNN

Now that you understand the structure and general flow of information in an RNN, let's get more specific and compute the actual activations of the hidden layers, as well as the output layer. For simplicity, we will consider just a single hidden layer; however, the same concept applies to multilayer RNNs.

Each directed edge (the connections between boxes) in the representation of an RNN that we just looked at is associated with a weight matrix. Those weights do not depend on time, t; therefore, they are shared across the time axis. The different weight matrices in a single-layer RNN are as follows:

- $\boldsymbol{W}_{xh}$: The weight matrix between the input, $\boldsymbol{x}^{(t)}$, and the hidden layer, $\boldsymbol{h}$
- $\boldsymbol{W}_{hh}$: The weight matrix associated with the recurrent edge
- $\boldsymbol{W}_{ho}$: The weight matrix between the hidden layer and output layer

These weight matrices are depicted in *Figure 15.5*:

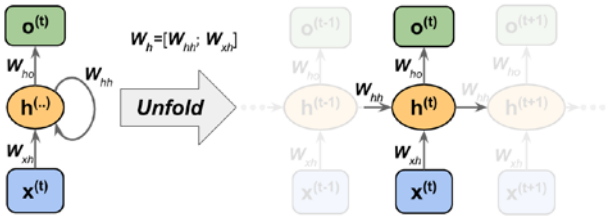

Figure 15.5: Applying weights to a single-layer RNN

In certain implementations, you may observe that the weight matrices, W_{xh} and W_{hh}, are concatenated to a combined matrix, $W_h = [W_{xh}; W_{hh}]$. Later in this section, we will make use of this notation as well.

Computing the activations is very similar to standard multilayer perceptrons and other types of feed-forward NNs. For the hidden layer, the net input, z_h (preactivation), is computed through a linear combination; that is, we compute the sum of the multiplications of the weight matrices with the corresponding vectors and add the bias unit:

$$\boldsymbol{z}_h^{(t)} = \boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h$$

Then, the activations of the hidden units at the time step, t, are calculated as follows:

$$\boldsymbol{h}^{(t)} = \sigma_h\left(\boldsymbol{z}_h^{(t)}\right) = \sigma_h\left(\boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h\right)$$

Here, $\boldsymbol{b}_h$ is the bias vector for the hidden units and $\sigma(\cdot)$ is the activation function of the hidden layer.

In case you want to use the concatenated weight matrix, $W_h = [W_{xh}; W_{hh}]$, the formula for computing hidden units will change, as follows:

$$\boldsymbol{h}^{(t)} = \sigma_h\left([\boldsymbol{W}_{xh}; \boldsymbol{W}_{hh}]\begin{bmatrix}\boldsymbol{x}^{(t)} \\ \boldsymbol{h}^{(t-1)}\end{bmatrix} + \boldsymbol{b}_h\right)$$

Once the activations of the hidden units at the current time step are computed, then the activations of the output units will be computed, as follows:

$$\boldsymbol{o}^{(t)} = \sigma_0\left(\boldsymbol{W}_{ho}\boldsymbol{h}^{(t)} + \boldsymbol{b}_0\right)$$

To help clarify this further, *Figure 15.6* shows the process of computing these activations with both formulations:

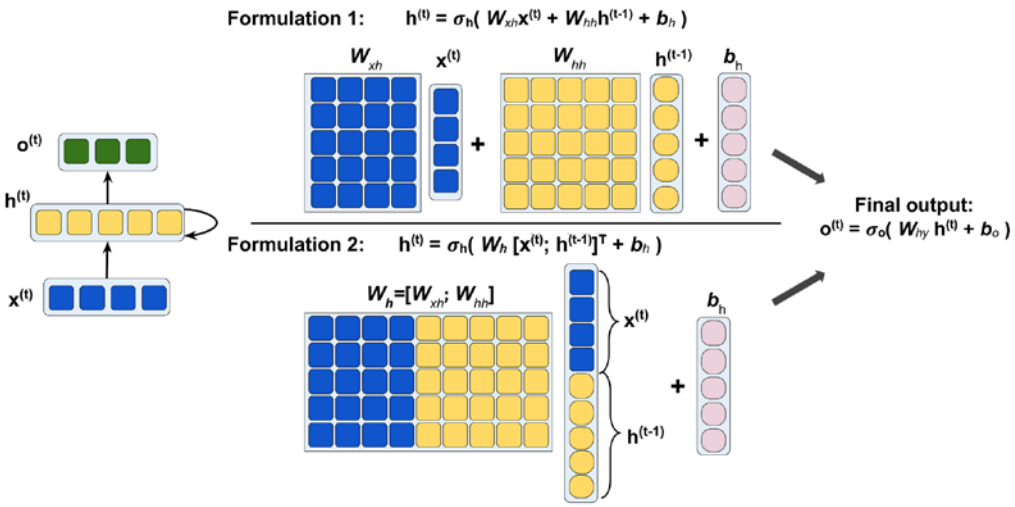

Figure 15.6: Computing the activations

Training RNNs using backpropagation through time (BPTT)

The learning algorithm for RNNs was introduced in 1990: *Backpropagation Through Time: What It Does and How to Do It* (*Paul Werbos*, *Proceedings of IEEE*, 78(10): 1550-1560, 1990).

The derivation of the gradients might be a bit complicated, but the basic idea is that the overall loss, L, is the sum of all the loss functions at times $t = 1$ to $t = T$:

$$L = \sum_{t=1}^{T} L^{(t)}$$

Since the loss at time t is dependent on the hidden units at all previous time steps $1 : t$, the gradient will be computed as follows:

$$\frac{\partial L^{(t)}}{\partial \boldsymbol{W}_{hh}} = \frac{\partial L^{(t)}}{\partial \boldsymbol{o}^{(t)}} \times \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \times \left(\sum_{k=1}^{t} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} \times \frac{\partial \boldsymbol{h}^{(k)}}{\partial \boldsymbol{W}_{hh}} \right)$$

Here, $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}$ is computed as a multiplication of adjacent time steps:

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}}$$

Hidden recurrence versus output recurrence

So far, you have seen recurrent networks in which the hidden layer has the recurrent property. However, note that there is an alternative model in which the recurrent connection comes from the output layer. In this case, the net activations from the output layer at the previous time step, $\boldsymbol{o}^{t-1}$, can be added in one of two ways:

- To the hidden layer at the current time step, $\boldsymbol{h}^{t}$ (shown in *Figure 15.7* as output-to-hidden recurrence)
- To the output layer at the current time step, $\boldsymbol{o}^{t}$ (shown in *Figure 15.7* as output-to-output recurrence)

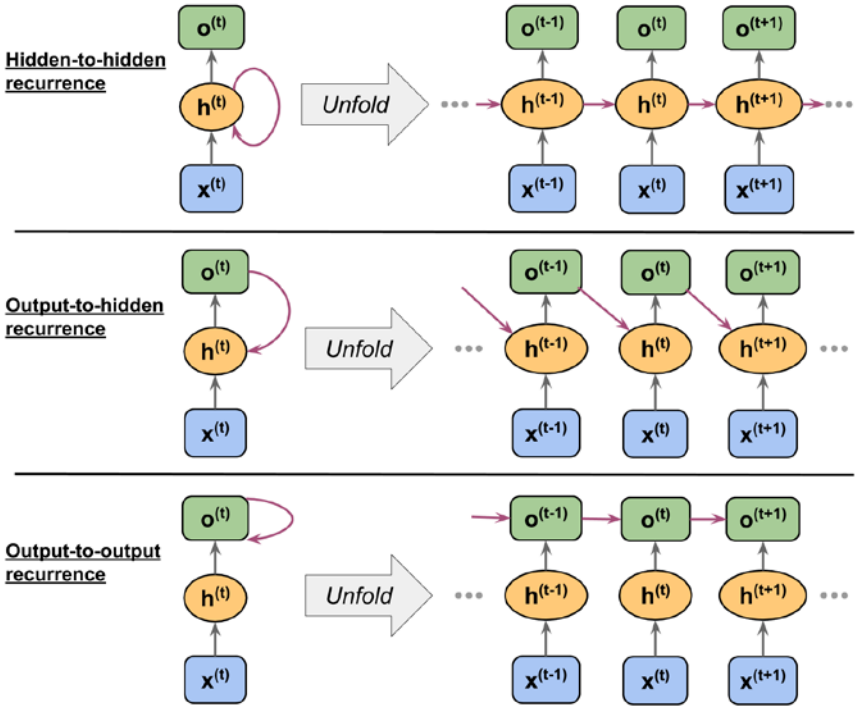

Figure 15.7: Different recurrent connection models

As shown in *Figure 15.7*, the differences between these architectures can be clearly seen in the recurring connections. Following our notation, the weights associated with the recurrent connection will be denoted for the hidden-to-hidden recurrence by $\boldsymbol{W}_{hh}$, for the output-to-hidden recurrence by $\boldsymbol{W}_{oh}$, and for the output-to-output recurrence by $\boldsymbol{W}_{oo}$. In some articles in literature, the weights associated with the recurrent connections are also denoted by $\boldsymbol{W}_{rec}$.

To see how this works in practice, let's manually compute the forward pass for one of these recurrent types. Using the `torch.nn` module, a recurrent layer can be defined via `RNN`, which is similar to the hidden-to-hidden recurrence. In the following code, we will create a recurrent layer from `RNN` and perform a forward pass on an input sequence of length 3 to compute the output. We will also manually compute the forward pass and compare the results with those of `RNN`.

First, let's create the layer and assign the weights and biases for our manual computations:

```
>>> import torch
>>> import torch.nn as nn
>>> torch.manual_seed(1)
>>> rnn_layer = nn.RNN(input_size=5, hidden_size=2,
...                    num_layers=1, batch_first=True)
>>> w_xh = rnn_layer.weight_ih_l0
>>> w_hh = rnn_layer.weight_hh_l0
>>> b_xh = rnn_layer.bias_ih_l0
>>> b_hh = rnn_layer.bias_hh_l0
>>> print('W_xh shape:', w_xh.shape)
>>> print('W_hh shape:', w_hh.shape)
>>> print('b_xh shape:', b_xh.shape)
>>> print('b_hh shape:', b_hh.shape)
W_xh shape: torch.Size([2, 5])
W_hh shape: torch.Size([2, 2])
b_xh shape: torch.Size([2])
b_hh shape: torch.Size([2])
```

The input shape for this layer is `(batch_size, sequence_length, 5)`, where the first dimension is the batch dimension (as we set `batch_first=True`), the second dimension corresponds to the sequence, and the last dimension corresponds to the features. Notice that we will output a sequence, which, for an input sequence of length 3, will result in the output sequence $\langle \boldsymbol{o}^{(0)}, \boldsymbol{o}^{(1)}, \boldsymbol{o}^{(2)} \rangle$. Also, `RNN` uses one layer by default, and you can set `num_layers` to stack multiple RNN layers together to form a stacked RNN.

Now, we will call the forward pass on the `rnn_layer` and manually compute the outputs at each time step and compare them:

```
>>> x_seq = torch.tensor([[1.0]*5, [2.0]*5, [3.0]*5]).float()
>>> ## output of the simple RNN:
>>> output, hn = rnn_layer(torch.reshape(x_seq, (1, 3, 5)))
>>> ## manually computing the output:
>>> out_man = []
>>> for t in range(3):
...     xt = torch.reshape(x_seq[t], (1, 5))
...     print(f'Time step {t} =>')
...     print('   Input           :', xt.numpy())
...
...     ht = torch.matmul(xt, torch.transpose(w_xh, 0, 1)) + b_hh
...     print('   Hidden          :', ht.detach().numpy()
...
```

```
...     if t > 0:
...         prev_h = out_man[t-1]
...     else:
...         prev_h = torch.zeros((ht.shape))
...     ot = ht + torch.matmul(prev_h, torch.transpose(w_hh, 0, 1)) \
...             + b_hh
...     ot = torch.tanh(ot)
...     out_man.append(ot)
...     print('   Output (manual) :', ot.detach().numpy())
...     print('   RNN output      :', output[:, t].detach().numpy())
...     print()
Time step 0 =>
   Input           : [[1. 1. 1. 1. 1.]]
   Hidden          : [[-0.4701929  0.5863904]]
   Output (manual) : [[-0.3519801   0.52525216]]
   RNN output      : [[-0.3519801   0.52525216]]

Time step 1 =>
   Input           : [[2. 2. 2. 2. 2.]]
   Hidden          : [[-0.88883156  1.2364397 ]]
   Output (manual) : [[-0.68424344  0.76074266]]
   RNN output      : [[-0.68424344  0.76074266]]

Time step 2 =>
   Input           : [[3. 3. 3. 3. 3.]]
   Hidden          : [[-1.3074701  1.886489 ]]
   Output (manual) : [[-0.8649416   0.90466356]]
   RNN output      : [[-0.8649416   0.90466356]]
```

In our manual forward computation, we used the hyperbolic tangent (tanh) activation function since it is also used in `RNN` (the default activation). As you can see from the printed results, the outputs from the manual forward computations exactly match the output of the `RNN` layer at each time step. Hopefully, this hands-on task has enlightened you on the mysteries of recurrent networks.

The challenges of learning long-range interactions

BPTT, which was briefly mentioned earlier, introduces some new challenges. Because of the multiplicative factor, $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}$, in computing the gradients of a loss function, the so-called **vanishing** and **exploding** gradient problems arise.

These problems are explained by the examples in *Figure 15.8*, which shows an RNN with only one hidden unit for simplicity:

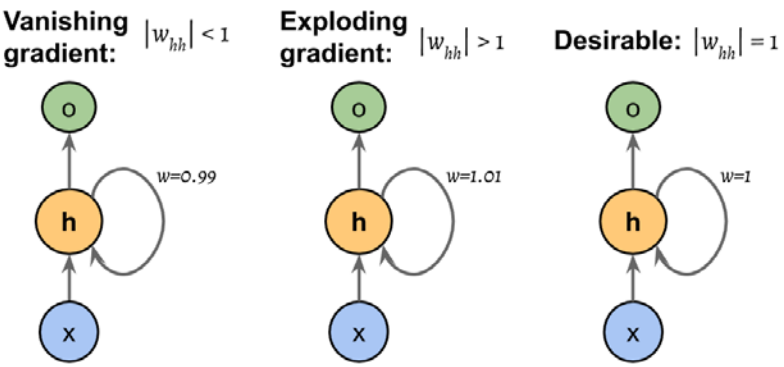

Figure 15.8: Problems in computing the gradients of the loss function

Basically, $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}$ has $t - k$ multiplications; therefore, multiplying the weight, w, by itself $t - k$ times results in a factor, w^{t-k}. As a result, if $|w| < 1$, this factor becomes very small when $t - k$ is large. On the other hand, if the weight of the recurrent edge is $|w| > 1$, then w^{t-k} becomes very large when $t - k$ is large. Note that a large $t - k$ refers to long-range dependencies. We can see that a naive solution to avoid vanishing or exploding gradients can be reached by ensuring $|w| = 1$. If you are interested and would like to investigate this in more detail, read *On the difficulty of training recurrent neural networks* by *R. Pascanu*, *T. Mikolov*, and *Y. Bengio*, 2012 (`https://arxiv.org/pdf/1211.5063.pdf`).

In practice, there are at least three solutions to this problem:

- Gradient clipping
- **Truncated backpropagation through time** (**TBPTT**)
- LSTM

Using gradient clipping, we specify a cut-off or threshold value for the gradients, and we assign this cut-off value to gradient values that exceed this value. In contrast, TBPTT simply limits the number of time steps that the signal can backpropagate after each forward pass. For example, even if the sequence has 100 elements or steps, we may only backpropagate the most recent 20 time steps.

While both gradient clipping and TBPTT can solve the exploding gradient problem, the truncation limits the number of steps that the gradient can effectively flow back and properly update the weights. On the other hand, LSTM, designed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber, has been more successful in vanishing and exploding gradient problems while modeling long-range dependencies through the use of memory cells. Let's discuss LSTM in more detail.

Long short-term memory cells

As stated previously, LSTMs were first introduced to overcome the vanishing gradient problem (*Long Short-Term Memory* by *S. Hochreiter* and *J. Schmidhuber*, *Neural Computation*, 9(8): 1735-1780, 1997). The building block of an LSTM is a **memory cell**, which essentially represents or replaces the hidden layer of standard RNNs.

In each memory cell, there is a recurrent edge that has the desirable weight, $w = 1$, as we discussed, to overcome the vanishing and exploding gradient problems. The values associated with this recurrent edge are collectively called the **cell state**. The unfolded structure of a modern LSTM cell is shown in *Figure 15.9*:

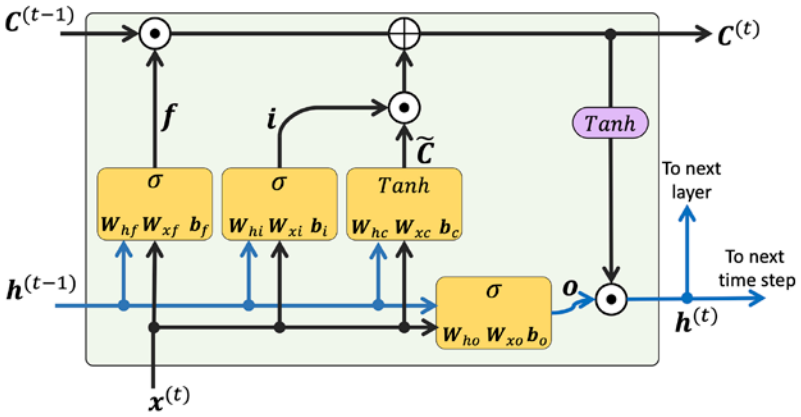

Figure 15.9: The structure of an LSTM cell

Notice that the cell state from the previous time step, $C^{(t-1)}$, is modified to get the cell state at the current time step, $C^{(t)}$, without being multiplied directly by any weight factor. The flow of information in this memory cell is controlled by several computation units (often called *gates*) that will be described here. In the figure, $\odot$ refers to the **element-wise product** (element-wise multiplication) and $\oplus$ means **element-wise summation** (element-wise addition). Furthermore, $\boldsymbol{x}^{(t)}$ refers to the input data at time t, and $\boldsymbol{h}^{(t-1)}$ indicates the hidden units at time $t - 1$. Four boxes are indicated with an activation function, either the sigmoid function (σ) or tanh, and a set of weights; these boxes apply a linear combination by performing matrix-vector multiplications on their inputs (which are $\boldsymbol{h}^{(t-1)}$ and $\boldsymbol{x}^{(t)}$). These units of computation with sigmoid activation functions, whose output units are passed through $\odot$, are called gates.

In an LSTM cell, there are three different types of gates, which are known as the forget gate, the input gate, and the output gate:

The **forget gate** (f_t) allows the memory cell to reset the cell state without growing indefinitely. In fact, the forget gate decides which information is allowed to go through and which information to suppress. Now, $\boldsymbol{f}_t$ is computed as follows:

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_{xf}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hf}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_f\right)$$

Note that the forget gate was not part of the original LSTM cell; it was added a few years later to improve the original model (*Learning to Forget: Continual Prediction with LSTM* by *F. Gers*, *J. Schmidhuber*, and *F. Cummins*, *Neural Computation 12*, 2451-2471, 2000).

The **input gate** ($\boldsymbol{i}_t$) and **candidate value** ($\widetilde{\boldsymbol{C}}_t$) are responsible for updating the cell state. They are computed as follows:

$$\begin{aligned}\boldsymbol{i}_t &= \sigma\left(\boldsymbol{W}_{xi}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hi}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i\right)\\ \widetilde{\boldsymbol{C}}_t &= \tanh\left(\boldsymbol{W}_{xc}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hc}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_c\right)\end{aligned}$$

The cell state at time t is computed as follows:

$$\boldsymbol{C}^{(t)} = \left(\boldsymbol{C}^{(t-1)} \odot \boldsymbol{f}_t\right) \oplus \left(\boldsymbol{i}_t \odot \tilde{C}_t\right)$$

The **output gate** ($\boldsymbol{o}_t$) decides how to update the values of hidden units:

$$\boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_{xo}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{ho}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_o\right)$$

Given this, the hidden units at the current time step are computed as follows:

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}_t \odot \tanh\left(\boldsymbol{C}^{(t)}\right)$$

The structure of an LSTM cell and its underlying computations might seem very complex and hard to implement. However, the good news is that PyTorch has already implemented everything in optimized wrapper functions, which allows us to define our LSTM cells easily and efficiently. We will apply RNNs and LSTMs to real-world datasets later in this chapter.

Other advanced RNN models

LSTMs provide a basic approach for modeling long-range dependencies in sequences. Yet, it is important to note that there are many variations of LSTMs described in literature (*An Empirical Exploration of Recurrent Network Architectures* by *Rafal Jozefowicz*, *Wojciech Zaremba*, and *Ilya Sutskever*, *Proceedings of ICML*, 2342-2350, 2015). Also worth noting is a more recent approach, **gated recurrent unit** (**GRU**), which was proposed in 2014. GRUs have a simpler architecture than LSTMs; therefore, they are computationally more efficient, while their performance in some tasks, such as polyphonic music modeling, is comparable to LSTMs. If you are interested in learning more about these modern RNN architectures, refer to the paper, *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling* by *Junyoung Chung* and others, 2014 (`https://arxiv.org/pdf/1412.3555v1.pdf`).

Implementing RNNs for sequence modeling in PyTorch

Now that we have covered the underlying theory behind RNNs, we are ready to move on to the more practical portion of this chapter: implementing RNNs in PyTorch. During the rest of this chapter, we will apply RNNs to two common problem tasks:

1. Sentiment analysis
2. Language modeling

These two projects, which we will walk through together in the following pages, are both fascinating but also quite involved. Thus, instead of providing the code all at once, we will break the implementation up into several steps and discuss the code in detail. If you like to have a big picture overview and want to see all the code at once before diving into the discussion, take a look at the code implementation first.

Project one – predicting the sentiment of IMDb movie reviews

You may recall from *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*, that sentiment analysis is concerned with analyzing the expressed opinion of a sentence or a text document. In this section and the following subsections, we will implement a multilayer RNN for sentiment analysis using a many-to-one architecture.

In the next section, we will implement a many-to-many RNN for an application of language modeling. While the chosen examples are purposefully simple to introduce the main concepts of RNNs, language modeling has a wide range of interesting applications, such as building chatbots—giving computers the ability to directly talk and interact with humans.

Preparing the movie review data

In *Chapter 8*, we preprocessed and cleaned the review dataset. And we will do the same now. First, we will import the necessary modules and read the data from `torchtext` (which we will install via `pip install torchtext`; version 0.10.0 was used as of late 2021) as follows:

```
>>> from torchtext.datasets import IMDB
>>> train_dataset = IMDB(split='train')
>>> test_dataset = IMDB(split='test')
```

Each set has 25,000 samples. And each sample of the datasets consists of two elements, the sentiment label representing the target label we want to predict (`neg` refers to negative sentiment and `pos` refers to positive sentiment), and the movie review text (the input features). The text component of these movie reviews is sequences of words, and the RNN model classifies each sequence as a positive (`1`) or negative (`0`) review.

However, before we can feed the data into an RNN model, we need to apply several preprocessing steps:

1. Split the training dataset into separate training and validation partitions.
2. Identify the unique words in the training dataset
3. Map each unique word to a unique integer and encode the review text into encoded integers (an index of each unique word)
4. Divide the dataset into mini-batches as input to the model

Let's proceed with the first step: creating a training and validation partition from the `train_dataset` we read earlier:

```
>>> ## Step 1: create the datasets
>>> from torch.utils.data.dataset import random_split
>>> torch.manual_seed(1)
>>> train_dataset, valid_dataset = random_split(
...     list(train_dataset), [20000, 5000])
```

The original training dataset contains 25,000 examples. 20,000 examples are randomly chosen for training, and 5,000 for validation.

To prepare the data for input to an NN, we need to encode it into numeric values, as was mentioned in *steps 2* and *3*. To do this, we will first find the unique words (tokens) in the training dataset. While finding unique tokens is a process for which we can use Python datasets, it can be more efficient to use the `Counter` class from the `collections` package, which is part of Python's standard library.

In the following code, we will instantiate a new `Counter` object (`token_counts`) that will collect the unique word frequencies. Note that in this particular application (and in contrast to the bag-of-words model), we are only interested in the set of unique words and won't require the word counts, which are created as a side product. To split the text into words (or tokens), we will reuse the `tokenizer` function we developed in *Chapter 8*, which also removes HTML markups as well as punctuation and other non-letter characters:

The code for collecting unique tokens is as follows:

```
>>> ## Step 2: find unique tokens (words)
>>> import re
>>> from collections import Counter, OrderedDict
>>>
>>> def tokenizer(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall(
...         '(?::|;|=)(?:-)?(?:\)|\(|D|P)', text.lower()
...     )
...     text = re.sub('[\W]+', ' ', text.lower()) +\
...         ' '.join(emoticons).replace('-', '')
...     tokenized = text.split()
```

```
...     return tokenized
>>>
>>> token_counts = Counter()
>>> for label, line in train_dataset:
...     tokens = tokenizer(line)
...     token_counts.update(tokens)
>>> print('Vocab-size:', len(token_counts))
Vocab-size: 69023
```

If you want to learn more about `Counter`, refer to its documentation at `https://docs.python.org/3/library/collections.html#collections.Counter`.

Next, we are going to map each unique word to a unique integer. This can be done manually using a Python dictionary, where the keys are the unique tokens (words) and the value associated with each key is a unique integer. However, the `torchtext` package already provides a class, `Vocab`, which we can use to create such a mapping and encode the entire dataset. First, we will create a `vocab` object by passing the ordered dictionary mapping tokens to their corresponding occurrence frequencies (the ordered dictionary is the sorted `token_counts`). Second, we will prepend two special tokens to the vocabulary – the padding and the unknown token:

```
>>> ## Step 3: encoding each unique token into integers
>>> from torchtext.vocab import vocab
>>> sorted_by_freq_tuples = sorted(
...     token_counts.items(), key=lambda x: x[1], reverse=True
... )
>>> ordered_dict = OrderedDict(sorted_by_freq_tuples)
>>> vocab = vocab(ordered_dict)
>>> vocab.insert_token("<pad>", 0)
>>> vocab.insert_token("<unk>", 1)
>>> vocab.set_default_index(1)
```

To demonstrate how to use the `vocab` object, we will convert an example input text into a list of integer values:

```
>>> print([vocab[token] for token in ['this', 'is',
...     'an', 'example']])
[11, 7, 35, 457]
```

Note that there might be some tokens in the validation or testing data that are not present in the training data and are thus not included in the mapping. If we have q tokens (that is, the size of `token_counts` passed to `Vocab`, which in this case is 69,023), then all tokens that haven't been seen before, and are thus not included in `token_counts`, will be assigned the integer 1 (a placeholder for the unknown token). In other words, the index 1 is reserved for unknown words. Another reserved value is the integer 0, which serves as a placeholder, a so-called *padding token*, for adjusting the sequence length. Later, when we are building an RNN model in PyTorch, we will consider this placeholder, 0, in more detail.

We can define the `text_pipeline` function to transform each text in the dataset accordingly and the `label_pipeline` function to convert each label to 1 or 0:

```
>>> ## Step 3-A: define the functions for transformation
>>> text_pipeline =\
...     lambda x: [vocab[token] for token in tokenizer(x)]
>>> label_pipeline = lambda x: 1. if x == 'pos' else 0.
```

We will generate batches of samples using `DataLoader` and pass the data processing pipelines declared previously to the argument `collate_fn`. We will wrap the text encoding and label transformation function into the `collate_batch` function:

```
>>> ## Step 3-B: wrap the encode and transformation function
... def collate_batch(batch):
...     label_list, text_list, lengths = [], [], []
...     for _label, _text in batch:
...         label_list.append(label_pipeline(_label))
...         processed_text = torch.tensor(text_pipeline(_text),
...                                       dtype=torch.int64)
...         text_list.append(processed_text)
...         lengths.append(processed_text.size(0))
...     label_list = torch.tensor(label_list)
...     lengths = torch.tensor(lengths)
...     padded_text_list = nn.utils.rnn.pad_sequence(
...         text_list, batch_first=True)
...     return padded_text_list, label_list, lengths
>>>
>>> ## Take a small batch
>>> from torch.utils.data import DataLoader
>>> dataloader = DataLoader(train_dataset, batch_size=4,
...                         shuffle=False, collate_fn=collate_batch)
```

So far, we've converted sequences of words into sequences of integers, and labels of `pos` or `neg` into 1 or 0. However, there is one issue that we need to resolve—the sequences currently have different lengths (as shown in the result of executing the following code for four examples). Although, in general, RNNs can handle sequences with different lengths, we still need to make sure that all the sequences in a mini-batch have the same length to store them efficiently in a tensor.

PyTorch provides an efficient method, `pad_sequence()`, which will automatically pad the consecutive elements that are to be combined into a batch with placeholder values (0s) so that all sequences within a batch will have the same shape. In the previous code, we already created a data loader of a small batch size from the training dataset and applied the `collate_batch` function, which itself included a `pad_sequence()` call.

However, to illustrate how padding works, we will take the first batch and print the sizes of the individual elements before combining these into mini-batches, as well as the dimensions of the resulting mini-batches:

```
>>> text_batch, label_batch, length_batch = next(iter(dataloader))
>>> print(text_batch)
tensor([[   35,  1742,     7,   449,   723,     6,   302,     4,
...
0,     0,     0,     0,     0,     0,     0,     0]],
>>> print(label_batch)
tensor([1., 1., 1., 0.])
>>> print(length_batch)
tensor([165,  86, 218, 145])
>>> print(text_batch.shape)
torch.Size([4, 218])
```

As you can observe from the printed tensor shapes, the number of columns in the first batch is 218, which resulted from combining the first four examples into a single batch and using the maximum size of these examples. This means that the other three examples (whose lengths are 165, 86, and 145, respectively) in this batch are padded as much as necessary to match this size.

Finally, let's divide all three datasets into data loaders with a batch size of 32:

```
>>> batch_size = 32
>>> train_dl = DataLoader(train_dataset, batch_size=batch_size,
...                       shuffle=True, collate_fn=collate_batch)
>>> valid_dl = DataLoader(valid_dataset, batch_size=batch_size,
...                       shuffle=False, collate_fn=collate_batch)
>>> test_dl = DataLoader(test_dataset, batch_size=batch_size,
...                      shuffle=False, collate_fn=collate_batch)
```

Now, the data is in a suitable format for an RNN model, which we are going to implement in the following subsections. In the next subsection, however, we will first discuss feature **embedding**, which is an optional but highly recommended preprocessing step that is used to reduce the dimensionality of the word vectors.

Embedding layers for sentence encoding

During the data preparation in the previous step, we generated sequences of the same length. The elements of these sequences were integer numbers that corresponded to the *indices* of unique words. These word indices can be converted into input features in several different ways. One naive way is to apply one-hot encoding to convert the indices into vectors of zeros and ones. Then, each word will be mapped to a vector whose size is the number of unique words in the entire dataset. Given that the number of unique words (the size of the vocabulary) can be in the order of 10^4 – 10^5, which will also be the number of our input features, a model trained on such features may suffer from the **curse of dimensionality**. Furthermore, these features are very sparse since all are zero except one.

A more elegant approach is to map each word to a vector of a fixed size with real-valued elements (not necessarily integers). In contrast to the one-hot encoded vectors, we can use finite-sized vectors to represent an infinite number of real numbers. (In theory, we can extract infinite real numbers from a given interval, for example [–1, 1].)

This is the idea behind embedding, which is a feature-learning technique that we can utilize here to automatically learn the salient features to represent the words in our dataset. Given the number of unique words, n_{words}, we can select the size of the embedding vectors (a.k.a., embedding dimension) to be much smaller than the number of unique words ($embedding\_dim << n_{words}$) to represent the entire vocabulary as input features.

The advantages of embedding over one-hot encoding are as follows:

- A reduction in the dimensionality of the feature space to decrease the effect of the curse of dimensionality
- The extraction of salient features since the embedding layer in an NN can be optimized (or learned)

The following schematic representation shows how embedding works by mapping token indices to a trainable embedding matrix:

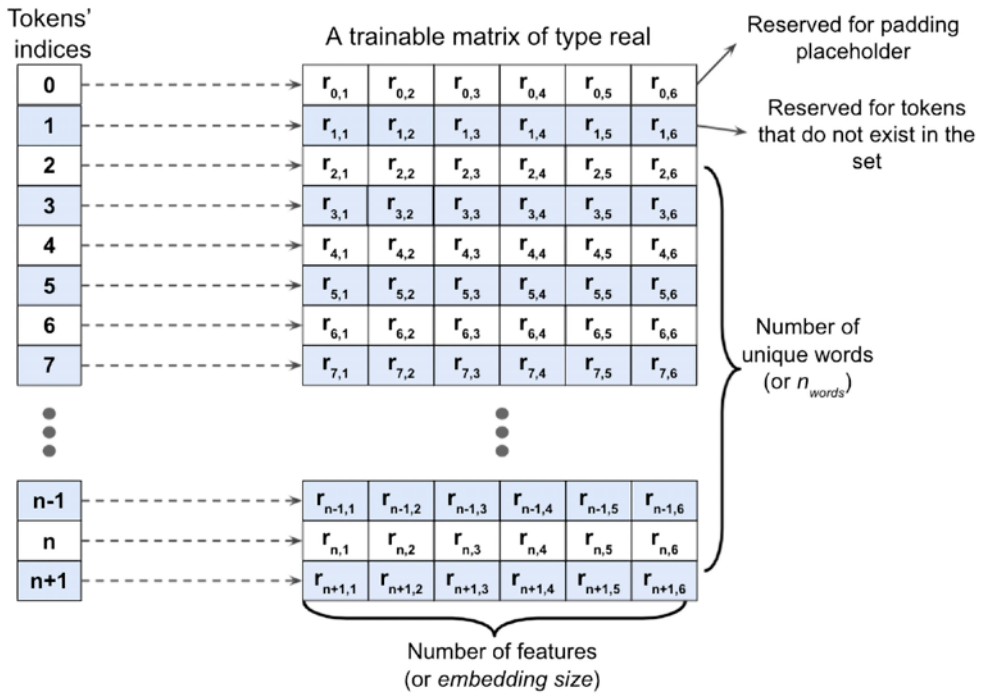

Figure 15.10: A breakdown of how embedding works

Given a set of tokens of size $n + 2$ (n is the size of the token set, plus index 0 is reserved for the padding placeholder, and 1 is for the words not present in the token set), an embedding matrix of size $(n + 2) \times embedding\_dim$ will be created where each row of this matrix represents numeric features associated with a token. Therefore, when an integer index, i, is given as input to the embedding, it will look up the corresponding row of the matrix at index i and return the numeric features. The embedding matrix serves as the input layer to our NN models. In practice, creating an embedding layer can simply be done using `nn.Embedding`. Let's see an example where we will create an embedding layer and apply it to a batch of two samples, as follows:

```
>>> embedding = nn.Embedding(
...     num_embeddings=10,
...     embedding_dim=3,
...     padding_idx=0)
>>> # a batch of 2 samples of 4 indices each
>>> text_encoded_input = torch.LongTensor([[1,2,4,5],[4,3,2,0]])
>>> print(embedding(text_encoded_input))
tensor([[[-0.7027,  0.3684, -0.5512],
         [-0.4147,  1.7891, -1.0674],
         [ 1.1400,  0.1595, -1.0167],
         [ 0.0573, -1.7568,  1.9067]],

        [[ 1.1400,  0.1595, -1.0167],
         [-0.8165, -0.0946, -0.1881],
         [-0.4147,  1.7891, -1.0674],
         [ 0.0000,  0.0000,  0.0000]]], grad_fn=<EmbeddingBackward>)
```

The input to this model (embedding layer) must have rank 2 with the dimensionality *batchsize* × *input_length*, where *input_length* is the length of sequences (here, 4). For example, an input sequence in the mini-batch could be <1, 5, 9, 2>, where each element of this sequence is the index of the unique words. The output will have the dimensionality *batchsize* × *input_length* × *embedding_dim*, where *embedding_dim* is the size of the embedding features (here, set to 3). The other argument provided to the embedding layer, `num_embeddings`, corresponds to the unique integer values that the model will receive as input (for instance, $n + 2$, set here to 10). Therefore, the embedding matrix in this case has the size 10×6.

`padding_idx` indicates the token index for padding (here, 0), which, if specified, will not contribute to the gradient updates during training. In our example, the length of the original sequence of the second sample is 3, and we padded it with 1 more element 0. The embedding output of the padded element is [0, 0, 0].

Building an RNN model

Now we're ready to build an RNN model. Using the `nn.Module` class, we can combine the embedding layer, the recurrent layers of the RNN, and the fully connected non-recurrent layers. For the recurrent layers, we can use any of the following implementations:

- `RNN`: a regular RNN layer, that is, a fully connected recurrent layer
- `LSTM`: a long short-term memory RNN, which is useful for capturing the long-term dependencies
- `GRU`: a recurrent layer with a gated recurrent unit, as proposed in *Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation* by *K. Cho* et al., 2014 (`https://arxiv.org/abs/1406.1078v3`), as an alternative to LSTMs

To see how a multilayer RNN model can be built using one of these recurrent layers, in the following example, we will create an RNN model with two recurrent layers of type `RNN`. Finally, we will add a non-recurrent fully connected layer as the output layer, which will return a single output value as the prediction:

```
>>> class RNN(nn.Module):
...     def __init__(self, input_size, hidden_size):
...         super().__init__()
...         self.rnn = nn.RNN(input_size, hidden_size, num_layers=2,
...                           batch_first=True)
...         # self.rnn = nn.GRU(input_size, hidden_size, num_layers,
...         #                   batch_first=True)
...         # self.rnn = nn.LSTM(input_size, hidden_size, num_layers,
...         #                    batch_first=True)
...         self.fc = nn.Linear(hidden_size, 1)
...
...     def forward(self, x):
...         _, hidden = self.rnn(x)
...         out = hidden[-1, :, :] # we use the final hidden state
...                                # from the last hidden layer as
...                                # the input to the fully connected
...                                # layer
...         out = self.fc(out)
...         return out
>>>
>>> model = RNN(64, 32)
>>> print(model)
>>> model(torch.randn(5, 3, 64))
RNN(
  (rnn): RNN(64, 32, num_layers=2, batch_first=True)
  (fc): Linear(in_features=32, out_features=1, bias=True)
)
```

```
tensor([[ 0.0010],
        [ 0.2478],
        [ 0.0573],
        [ 0.1637],
        [-0.0073]], grad_fn=<AddmmBackward>)
```

As you can see, building an RNN model using these recurrent layers is pretty straightforward. In the next subsection, we will go back to our sentiment analysis task and build an RNN model to solve that.

Building an RNN model for the sentiment analysis task

Since we have very long sequences, we are going to use an LSTM layer to account for long-range effects. We will create an RNN model for sentiment analysis, starting with an embedding layer producing word embeddings of feature size 20 (`embed_dim=20`). Then, a recurrent layer of type LSTM will be added. Finally, we will add a fully connected layer as a hidden layer and another fully connected layer as the output layer, which will return a single class-membership probability value via the logistic sigmoid activation as the prediction:

```
>>> class RNN(nn.Module):
...     def __init__(self, vocab_size, embed_dim, rnn_hidden_size,
...                  fc_hidden_size):
...         super().__init__()
...         self.embedding = nn.Embedding(vocab_size,
...                                       embed_dim,
...                                       padding_idx=0)
...         self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
...                            batch_first=True)
...         self.fc1 = nn.Linear(rnn_hidden_size, fc_hidden_size)
...         self.relu = nn.ReLU()
...         self.fc2 = nn.Linear(fc_hidden_size, 1)
...         self.sigmoid = nn.Sigmoid()
...
...     def forward(self, text, lengths):
...         out = self.embedding(text)
...         out = nn.utils.rnn.pack_padded_sequence(
...             out, lengths.cpu().numpy(), enforce_sorted=False, batch_first=True
...         )
...         out, (hidden, cell) = self.rnn(out)
...         out = hidden[-1, :, :]
...         out = self.fc1(out)
...         out = self.relu(out)
...         out = self.fc2(out)
...         out = self.sigmoid(out)
...         return out
```

```
>>>
>>> vocab_size = len(vocab)
>>> embed_dim = 20
>>> rnn_hidden_size = 64
>>> fc_hidden_size = 64
>>> torch.manual_seed(1)
>>> model = RNN(vocab_size, embed_dim,
                rnn_hidden_size, fc_hidden_size)
>>> model
RNN(
  (embedding): Embedding(69025, 20, padding_idx=0)
  (rnn): LSTM(20, 64, batch_first=True)
  (fc1): Linear(in_features=64, out_features=64, bias=True)
  (relu): ReLU()
  (fc2): Linear(in_features=64, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

Now we will develop the `train` function to train the model on the given dataset for one epoch and return the classification accuracy and loss:

```
>>> def train(dataloader):
...     model.train()
...     total_acc, total_loss = 0, 0
...     for text_batch, label_batch, lengths in dataloader:
...         optimizer.zero_grad()
...         pred = model(text_batch, lengths)[:, 0]
...         loss = loss_fn(pred, label_batch)
...         loss.backward()
...         optimizer.step()
...         total_acc += (
...             (pred >= 0.5).float() == label_batch
...         ).float().sum().item()
...         total_loss += loss.item()*label_batch.size(0)
...     return total_acc/len(dataloader.dataset), \
...            total_loss/len(dataloader.dataset)
```

Similarly, we will develop the `evaluate` function to measure the model's performance on a given dataset:

```
>>> def evaluate(dataloader):
...     model.eval()
...     total_acc, total_loss = 0, 0
```

```
...     with torch.no_grad():
...         for text_batch, label_batch, lengths in dataloader:
...             pred = model(text_batch, lengths)[:, 0]
...             loss = loss_fn(pred, label_batch)
...             total_acc += (
...                 (pred>=0.5).float() == label_batch
...             ).float().sum().item()
...             total_loss += loss.item()*label_batch.size(0)
...     return total_acc/len(dataloader.dataset), \
...            total_loss/len(dataloader.dataset)
```

The next step is to create a loss function and optimizer (Adam optimizer). For a binary classification with a single class-membership probability output, we use the binary cross-entropy loss (`BCELoss`) as the loss function:

```
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Now we will train the model for 10 epochs and display the training and validation performances:

```
>>> num_epochs = 10
>>> torch.manual_seed(1)
>>> for epoch in range(num_epochs):
...     acc_train, loss_train = train(train_dl)
...     acc_valid, loss_valid = evaluate(valid_dl)
...     print(f'Epoch {epoch} accuracy: {acc_train:.4f}'
...           f' val_accuracy: {acc_valid:.4f}')
Epoch 0 accuracy: 0.5843 val_accuracy: 0.6240
Epoch 1 accuracy: 0.6364 val_accuracy: 0.6870
Epoch 2 accuracy: 0.8020 val_accuracy: 0.8194
Epoch 3 accuracy: 0.8730 val_accuracy: 0.8454
Epoch 4 accuracy: 0.9092 val_accuracy: 0.8598
Epoch 5 accuracy: 0.9347 val_accuracy: 0.8630
Epoch 6 accuracy: 0.9507 val_accuracy: 0.8636
Epoch 7 accuracy: 0.9655 val_accuracy: 0.8654
Epoch 8 accuracy: 0.9765 val_accuracy: 0.8528
Epoch 9 accuracy: 0.9839 val_accuracy: 0.8596
```

After training this model for 10 epochs, we will evaluate it on the test data:

```
>>> acc_test, _ = evaluate(test_dl)
>>> print(f'test_accuracy: {acc_test:.4f}')
test_accuracy: 0.8512
```

It showed 85 percent accuracy. (Note that this result is not the best when compared to the state-of-the-art methods used on the IMDb dataset. The goal was simply to show how an RNN works in PyTorch.)

More on the bidirectional RNN

In addition, we will set the `bidirectional` configuration of the `LSTM` to `True`, which will make the recurrent layer pass through the input sequences from both directions, start to end, as well as in the reverse direction:

```
>>> class RNN(nn.Module):
...     def __init__(self, vocab_size, embed_dim,
...                  rnn_hidden_size, fc_hidden_size):
...         super().__init__()
...         self.embedding = nn.Embedding(
...             vocab_size, embed_dim, padding_idx=0
...         )
...         self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
...                            batch_first=True, bidirectional=True)
...         self.fc1 = nn.Linear(rnn_hidden_size*2, fc_hidden_size)
...         self.relu = nn.ReLU()
...         self.fc2 = nn.Linear(fc_hidden_size, 1)
...         self.sigmoid = nn.Sigmoid()
...
...     def forward(self, text, lengths):
...         out = self.embedding(text)
...         out = nn.utils.rnn.pack_padded_sequence(
...             out, lengths.cpu().numpy(), enforce_sorted=False, batch_first=True
...         )
...         _, (hidden, cell) = self.rnn(out)
...         out = torch.cat((hidden[-2, :, :],
...                          hidden[-1, :, :]), dim=1)
...         out = self.fc1(out)
...         out = self.relu(out)
...         out = self.fc2(out)
...         out = self.sigmoid(out)
...         return out
>>>
>>> torch.manual_seed(1)
>>> model = RNN(vocab_size, embed_dim,
...             rnn_hidden_size, fc_hidden_size)
>>> model
```

```
RNN(
  (embedding): Embedding(69025, 20, padding_idx=0)
  (rnn): LSTM(20, 64, batch_first=True, bidirectional=True)
  (fc1): Linear(in_features=128, out_features=64, bias=True)
  (relu): ReLU()
  (fc2): Linear(in_features=64, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

The bidirectional RNN layer makes two passes over each input sequence: a forward pass and a reverse or backward pass (note that this is not to be confused with the forward and backward passes in the context of backpropagation). The resulting hidden states of these forward and backward passes are usually concatenated into a single hidden state. Other merge modes include summation, multiplication (multiplying the results of the two passes), and averaging (taking the average of the two).

We can also try other types of recurrent layers, such as the regular `RNN`. However, as it turns out, a model built with regular recurrent layers won't be able to reach a good predictive performance (even on the training data). For example, if you try replacing the bidirectional LSTM layer in the previous code with a unidirectional `nn.RNN` (instead of `nn.LSTM`) layer and train the model on full-length sequences, you may observe that the loss will not even decrease during training. The reason is that the sequences in this dataset are too long, so a model with an `RNN` layer cannot learn the long-term dependencies and may suffer from vanishing or exploding gradient problems.

Project two — character-level language modeling in PyTorch

Language modeling is a fascinating application that enables machines to perform human language-related tasks, such as generating English sentences. One of the interesting studies in this area is *Generating Text with Recurrent Neural Networks* by *Ilya Sutskever*, *James Martens*, and *Geoffrey E. Hinton*, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011 (`https://pdfs.semanticscholar.org/93c2/0e38c85b69fc2d2eb314b3c1217913f7db11.pdf`).

In the model that we will build now, the input is a text document, and our goal is to develop a model that can generate new text that is similar in style to the input document. Examples of such input are a book or a computer program in a specific programming language.

In character-level language modeling, the input is broken down into a sequence of characters that are fed into our network one character at a time. The network will process each new character in conjunction with the memory of the previously seen characters to predict the next one.

Figure 15.11 shows an example of character-level language modeling (note that EOS stands for "end of sequence"):

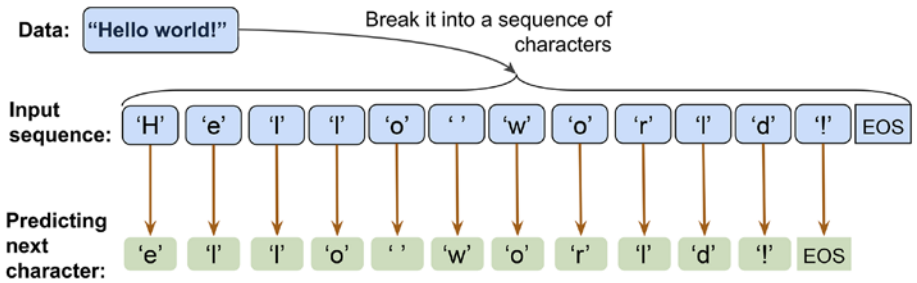

Figure 15.11: Character-level language modeling

We can break this implementation down into three separate steps: preparing the data, building the RNN model, and performing next-character prediction and sampling to generate new text.

Preprocessing the dataset

In this section, we will prepare the data for character-level language modeling.

To obtain the input data, visit the Project Gutenberg website at `https://www.gutenberg.org/`, which provides thousands of free e-books. For our example, you can download the book *The Mysterious Island*, by Jules Verne (published in 1874) in plain text format from `https://www.gutenberg.org/files/1268/1268-0.txt`.

Note that this link will take you directly to the download page. If you are using macOS or a Linux operating system, you can download the file with the following command in the terminal:

```
curl -O https://www.gutenberg.org/files/1268/1268-0.txt
```

If this resource becomes unavailable in the future, a copy of this text is also included in this chapter's code directory in the book's code repository at `https://github.com/rasbt/machine-learning-book`.

Once we have downloaded the dataset, we can read it into a Python session as plain text. Using the following code, we will read the text directly from the downloaded file and remove portions from the beginning and the end (these contain certain descriptions of the Gutenberg project). Then, we will create a Python variable, `char_set`, that represents the set of *unique* characters observed in this text:

```
>>> import numpy as np
>>> ## Reading and processing text
>>> with open('1268-0.txt', 'r', encoding="utf8") as fp:
...     text=fp.read()
>>> start_indx = text.find('THE MYSTERIOUS ISLAND')
>>> end_indx = text.find('End of the Project Gutenberg')
>>> text = text[start_indx:end_indx]
>>> char_set = set(text)
```

```
>>> print('Total Length:', len(text))
Total Length: 1112350
>>> print('Unique Characters:', len(char_set))
Unique Characters: 80
```

After downloading and preprocessing the text, we have a sequence consisting of 1,112,350 characters in total and 80 unique characters. However, most NN libraries and RNN implementations cannot deal with input data in string format, which is why we have to convert the text into a numeric format. To do this, we will create a simple Python dictionary that maps each character to an integer, `char2int`. We will also need a reverse mapping to convert the results of our model back to text. Although the reverse can be done using a dictionary that associates integer keys with character values, using a NumPy array and indexing the array to map indices to those unique characters is more efficient. *Figure 15.12* shows an example of converting characters into integers and the reverse for the words `"Hello"` and `"world"`:

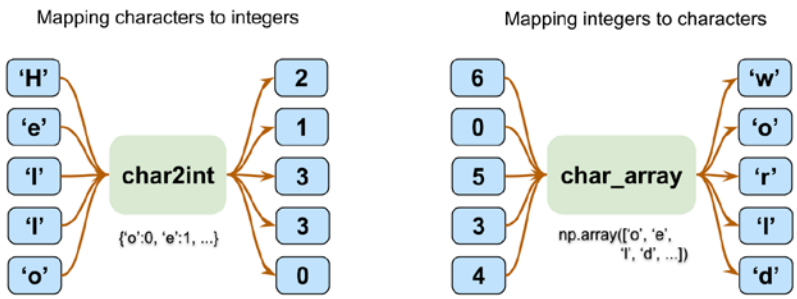

Figure 15.12: Character and integer mappings

Building the dictionary to map characters to integers, and reverse mapping via indexing a NumPy array, as was shown in the previous figure, is as follows:

```
>>> chars_sorted = sorted(char_set)
>>> char2int = {ch:i for i,ch in enumerate(chars_sorted)}
>>> char_array = np.array(chars_sorted)
>>> text_encoded = np.array(
...     [char2int[ch] for ch in text],
...     dtype=np.int32
... )
>>> print('Text encoded shape:', text_encoded.shape)
Text encoded shape: (1112350,)
>>> print(text[:15], '== Encoding ==>', text_encoded[:15])
>>> print(text_encoded[15:21], '== Reverse ==>',
...       ''.join(char_array[text_encoded[15:21]]))
THE MYSTERIOUS == Encoding ==> [44 32 29  1 37 48 43 44 29 42 33 39 45 43  1]
[33 43 36 25 38 28] == Reverse ==> ISLAND
```

The `text_encoded` NumPy array contains the encoded values for all the characters in the text. Now, we will print out the mappings of the first five characters from this array:

```
>>> for ex in text_encoded[:5]:
...     print('{} -> {}'.format(ex, char_array[ex]))
44 -> T
32 -> H
29 -> E
1 ->
37 -> M
```

Now, let's step back and look at the big picture of what we are trying to do. For the text generation task, we can formulate the problem as a classification task.

Suppose we have a set of sequences of text characters that are incomplete, as shown in *Figure 15.13*:

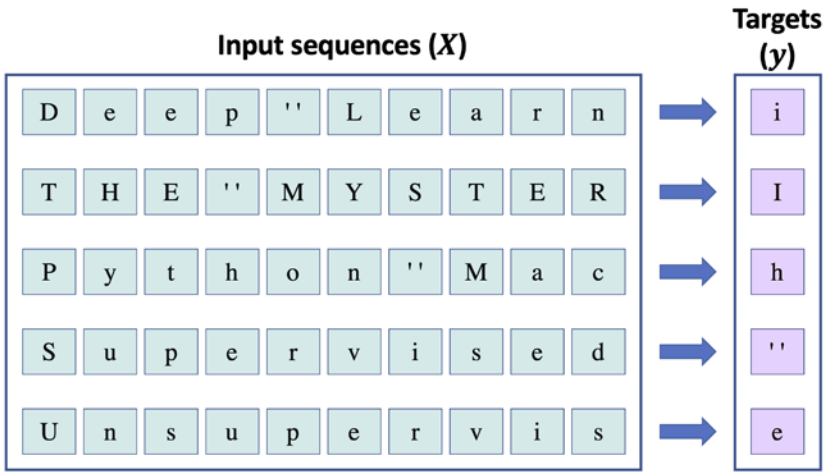

Figure 15.13: Predicting the next character for a text sequence

In *Figure 15.13*, we can consider the sequences shown in the left-hand box to be the input. In order to generate new text, our goal is to design a model that can predict the next character of a given input sequence, where the input sequence represents an incomplete text. For example, after seeing "Deep Learn," the model should predict "i" as the next character. Given that we have 80 unique characters, this problem becomes a multiclass classification task.

Starting with a sequence of length 1 (that is, one single letter), we can iteratively generate new text based on this multiclass classification approach, as illustrated in *Figure 15.14*:

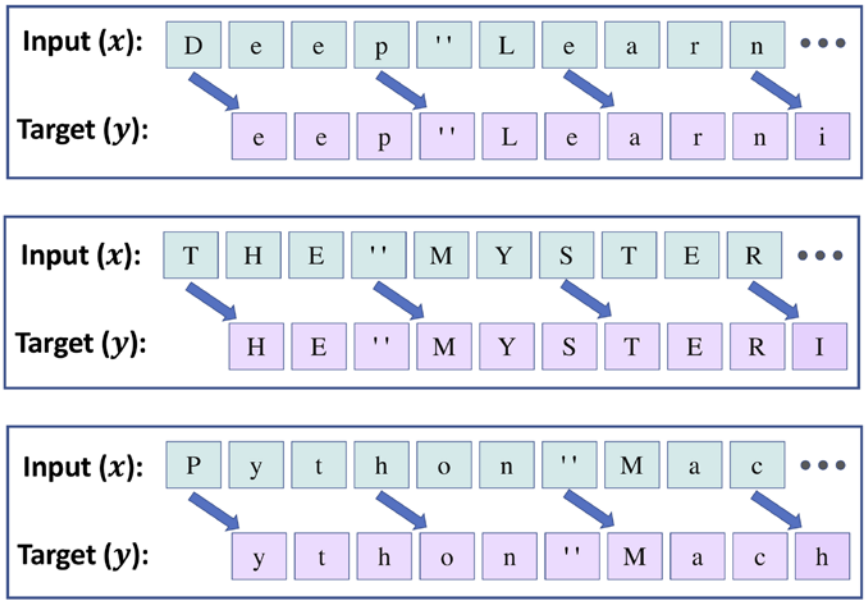

Figure 15.14: Generating next text based on this multiclass classification approach

To implement the text generation task in PyTorch, let's first clip the sequence length to 40. This means that the input tensor, x, consists of 40 tokens. In practice, the sequence length impacts the quality of the generated text. Longer sequences can result in more meaningful sentences. For shorter sequences, however, the model might focus on capturing individual words correctly, while ignoring the context for the most part. Although longer sequences usually result in more meaningful sentences, as mentioned, for long sequences, the RNN model will have problems capturing long-range dependencies. Thus, in practice, finding a sweet spot and good value for the sequence length is a hyperparameter optimization problem, which we have to evaluate empirically. Here, we are going to choose 40, as it offers a good trade-off.

As you can see in the previous figure, the inputs, x, and targets, y, are offset by one character. Hence, we will split the text into chunks of size 41: the first 40 characters will form the input sequence, x, and the last 40 elements will form the target sequence, y.

We have already stored the entire encoded text in its original order in `text_encoded`. We will first create text chunks consisting of 41 characters each. We will further get rid of the last chunk if it is shorter than 41 characters. As a result, the new chunked dataset, named `text_chunks`, will always contain sequences of size 41. The 41-character chunks will then be used to construct the sequence x (that is, the input), as well as the sequence y (that is, the target), both of which will have 40 elements. For instance, sequence x will consist of the elements with indices [0, 1, ..., 39]. Furthermore, since sequence y will be shifted by one position with respect to x, its corresponding indices will be [1, 2, ..., 40]. Then, we will transform the result into a `Dataset` object by applying a self-defined `Dataset` class:

```
>>> import torch
>>> from torch.utils.data import Dataset
>>> seq_length = 40
>>> chunk_size = seq_length + 1
>>> text_chunks = [text_encoded[i:i+chunk_size]
...                for i in range(len(text_encoded)-chunk_size)]
>>> from torch.utils.data import Dataset
>>> class TextDataset(Dataset):
...     def __init__(self, text_chunks):
...         self.text_chunks = text_chunks
...
...     def __len__(self):
...         return len(self.text_chunks)
...
...     def __getitem__(self, idx):
...         text_chunk = self.text_chunks[idx]
...         return text_chunk[:-1].long(), text_chunk[1:].long()
>>>
>>> seq_dataset = TextDataset(torch.tensor(text_chunks))
```

Let's take a look at some example sequences from this transformed dataset:

```
>>> for i, (seq, target) in enumerate(seq_dataset):
...     print(' Input (x): ',
...           repr(''.join(char_array[seq])))
...     print('Target (y): ',
...           repr(''.join(char_array[target])))
...     print()
...     if i == 1:
...         break
 Input (x): 'THE MYSTERIOUS ISLAND ***\n\n\n\n\nProduced b'
Target (y): 'HE MYSTERIOUS ISLAND ***\n\n\n\n\nProduced by'

 Input (x): 'HE MYSTERIOUS ISLAND ***\n\n\n\n\nProduced by'
Target (y): 'E MYSTERIOUS ISLAND ***\n\n\n\n\nProduced by '
```

Finally, the last step in preparing the dataset is to transform this dataset into mini-batches:

```
>>> from torch.utils.data import DataLoader
>>> batch_size = 64
>>> torch.manual_seed(1)
>>> seq_dl = DataLoader(seq_dataset, batch_size=batch_size,
...                     shuffle=True, drop_last=True)
```

Building a character-level RNN model

Now that the dataset is ready, building the model will be relatively straightforward:

```
>>> import torch.nn as nn
>>> class RNN(nn.Module):
...     def __init__(self, vocab_size, embed_dim, rnn_hidden_size):
...         super().__init__()
...         self.embedding = nn.Embedding(vocab_size, embed_dim)
...         self.rnn_hidden_size = rnn_hidden_size
...         self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
...                            batch_first=True)
...         self.fc = nn.Linear(rnn_hidden_size, vocab_size)
...
...     def forward(self, x, hidden, cell):
...         out = self.embedding(x).unsqueeze(1)
...         out, (hidden, cell) = self.rnn(out, (hidden, cell))
...         out = self.fc(out).reshape(out.size(0), -1)
...         return out, hidden, cell
...
...     def init_hidden(self, batch_size):
...         hidden = torch.zeros(1, batch_size, self.rnn_hidden_size)
...         cell = torch.zeros(1, batch_size, self.rnn_hidden_size)
...         return hidden, cell
```

Notice that we will need to have the logits as outputs of the model so that we can sample from the model predictions in order to generate new text. We will get to this sampling part later.

Then, we can specify the model parameters and create an RNN model:

```
>>> vocab_size = len(char_array)
>>> embed_dim = 256
>>> rnn_hidden_size = 512
>>> torch.manual_seed(1)
>>> model = RNN(vocab_size, embed_dim, rnn_hidden_size)
>>> model
RNN(
  (embedding): Embedding(80, 256)
```

```
  (rnn): LSTM(256, 512, batch_first=True)
  (fc): Linear(in_features=512, out_features=80, bias=True)
  (softmax): LogSoftmax(dim=1)
)
```

The next step is to create a loss function and optimizer (Adam optimizer). For a multiclass classification (we have `vocab_size=80` classes) with a single logits output for each target character, we use `CrossEntropyLoss` as the loss function:

```
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Now we will train the model for 10,000 epochs. In each epoch, we will use only one batch randomly chosen from the data loader, `seq_dl`. We will also display the training loss for every 500 epochs:

```
>>> num_epochs = 10000
>>> torch.manual_seed(1)
>>> for epoch in range(num_epochs):
...     hidden, cell = model.init_hidden(batch_size)
...     seq_batch, target_batch = next(iter(seq_dl))
...     optimizer.zero_grad()
...     loss = 0
...     for c in range(seq_length):
...         pred, hidden, cell = model(seq_batch[:, c], hidden, cell)
...         loss += loss_fn(pred, target_batch[:, c])
...     loss.backward()
...     optimizer.step()
...     loss = loss.item()/seq_length
...     if epoch % 500 == 0:
...         print(f'Epoch {epoch} loss: {loss:.4f}')
Epoch 0 loss: 1.9689
Epoch 500 loss: 1.4064
Epoch 1000 loss: 1.3155
Epoch 1500 loss: 1.2414
Epoch 2000 loss: 1.1697
Epoch 2500 loss: 1.1840
Epoch 3000 loss: 1.1469
Epoch 3500 loss: 1.1633
Epoch 4000 loss: 1.1788
Epoch 4500 loss: 1.0828
Epoch 5000 loss: 1.1164
Epoch 5500 loss: 1.0821
Epoch 6000 loss: 1.0764
```

```
Epoch 6500 loss: 1.0561
Epoch 7000 loss: 1.0631
Epoch 7500 loss: 0.9904
Epoch 8000 loss: 1.0053
Epoch 8500 loss: 1.0290
Epoch 9000 loss: 1.0133
Epoch 9500 loss: 1.0047
```

Next, we can evaluate the model to generate new text, starting with a given short string. In the next section, we will define a function to evaluate the trained model.

Evaluation phase – generating new text passages

The RNN model we trained in the previous section returns the logits of size 80 for each unique character. These logits can be readily converted to probabilities, via the softmax function, that a particular character will be encountered as the next character. To predict the next character in the sequence, we can simply select the element with the maximum logit value, which is equivalent to selecting the character with the highest probability. However, instead of always selecting the character with the highest likelihood, we want to (randomly) *sample* from the outputs; otherwise, the model will always produce the same text. PyTorch already provides a class, `torch.distributions.categorical.Categorical`, which we can use to draw random samples from a categorical distribution. To see how this works, let's generate some random samples from three categories [0, 1, 2], with input logits [1, 1, 1]:

```
>>> from torch.distributions.categorical import Categorical
>>> torch.manual_seed(1)
>>> logits = torch.tensor([[1.0, 1.0, 1.0]])
>>> print('Probabilities:',
...       nn.functional.softmax(logits, dim=1).numpy()[0])
Probabilities: [0.33333334 0.33333334 0.33333334]
>>> m = Categorical(logits=logits)
>>> samples = m.sample((10,))
>>> print(samples.numpy())
[[0]
 [0]
 [0]
 [0]
 [1]
 [0]
 [1]
 [2]
 [1]
 [1]]
```

As you can see, with the given logits, the categories have the same probabilities (that is, equiprobable categories). Therefore, if we use a large sample size (*num_samples* → ∞), we would expect the number of occurrences of each category to reach ≈ 1/3 of the sample size. If we change the logits to [1, 1, 3], then we would expect to observe more occurrences for category 2 (when a very large number of examples are drawn from this distribution):

```
>>> torch.manual_seed(1)
>>> logits = torch.tensor([[1.0, 1.0, 3.0]])
>>> print('Probabilities:', nn.functional.softmax(logits, dim=1).numpy()[0])
Probabilities: [0.10650698 0.10650698 0.78698605]
>>> m = Categorical(logits=logits)
>>> samples = m.sample((10,))
>>> print(samples.numpy())
[[0]
 [2]
 [2]
 [1]
 [2]
 [1]
 [2]
 [2]
 [2]
 [2]]
```

Using `Categorical`, we can generate examples based on the logits computed by our model.

We will define a function, `sample()`, that receives a short starting string, `starting_str`, and generate a new string, `generated_str`, which is initially set to the input string. `starting_str` is encoded to a sequence of integers, `encoded_input`. `encoded_input` is passed to the RNN model one character at a time to update the hidden states. The last character of `encoded_input` is passed to the model to generate a new character. Note that the output of the RNN model represents the logits (here, a vector of size 80, which is the total number of possible characters) for the next character after observing the input sequence by the model.

Here, we only use the `logits` output (that is, $\boldsymbol{o}^{(T)}$), which is passed to the `Categorical` class to generate a new sample. This new sample is converted to a character, which is then appended to the end of the generated string, `generated_text`, increasing its length by 1. Then, this process is repeated until the length of the generated string reaches the desired value. The process of consuming the generated sequence as input for generating new elements is called **autoregression**.

The code for the `sample()` function is as follows:

```
>>> def sample(model, starting_str,
...            len_generated_text=500,
...            scale_factor=1.0):
```

```
...     encoded_input = torch.tensor(
...         [char2int[s] for s in starting_str]
...     )
...     encoded_input = torch.reshape(
...         encoded_input, (1, -1)
...     )
...     generated_str = starting_str
...
...     model.eval()
...     hidden, cell = model.init_hidden(1)
...     for c in range(len(starting_str)-1):
...         _, hidden, cell = model(
...             encoded_input[:, c].view(1), hidden, cell
...         )
...
...     last_char = encoded_input[:, -1]
...     for i in range(len_generated_text):
...         logits, hidden, cell = model(
...             last_char.view(1), hidden, cell
...         )
...         logits = torch.squeeze(logits, 0)
...         scaled_logits = logits * scale_factor
...         m = Categorical(logits=scaled_logits)
...         last_char = m.sample()
...         generated_str += str(char_array[last_char])
...
...     return generated_str
```

Let's now generate some new text:

```
>>> torch.manual_seed(1)
>>> print(sample(model, starting_str='The island'))
The island had been made
and ovylore with think, captain?" asked Neb; "we do."

It was found, they full to time to remove. About this neur prowers, perhaps
ended? It is might be
rather rose?"

"Forward!" exclaimed Pencroft, "they were it? It seems to me?"

"The dog Top--"
```

```
"What can have been struggling sventy."

Pencroft calling, themselves in time to try them what proves that the sailor
and Neb bounded this tenarvan's feelings, and then
still hid head a grand furiously watched to the dorner nor his only
```

As you can see, the model generates mostly correct words, and, in some cases, the sentences are partially meaningful. You can further tune the training parameters, such as the length of input sequences for training, and the model architecture.

Furthermore, to control the predictability of the generated samples (that is, generating text following the learned patterns from the training text versus adding more randomness), the logits computed by the RNN model can be scaled before being passed to `Categorical` for sampling. The scaling factor, α, can be interpreted as an analog to the temperature in physics. Higher temperatures result in more entropy or randomness versus more predictable behavior at lower temperatures. By scaling the logits with $\alpha < 1$, the probabilities computed by the softmax function become more uniform, as shown in the following code:

```
>>> logits = torch.tensor([[1.0, 1.0, 3.0]])
>>> print('Probabilities before scaling:        ',
...       nn.functional.softmax(logits, dim=1).numpy()[0])
>>> print('Probabilities after scaling with 0.5:',
...       nn.functional.softmax(0.5*logits, dim=1).numpy()[0])
>>> print('Probabilities after scaling with 0.1:',
...       nn.functional.softmax(0.1*logits, dim=1).numpy()[0])
Probabilities before scaling:         [0.10650698 0.10650698 0.78698604]
Probabilities after scaling with 0.5: [0.21194156 0.21194156 0.57611688]
Probabilities after scaling with 0.1: [0.31042377 0.31042377 0.37915245]
```

As you can see, scaling the logits by $\alpha = 1$ results in near-uniform probabilities [0.31, 0.31, 0.38]. Now, we can compare the generated text with $\alpha = 2.0$ and $\alpha = 0.5$, as shown in the following points:

- $\alpha = 2.0 \rightarrow$ more predictable:

```
>>> torch.manual_seed(1)
>>> print(sample(model, starting_str='The island',
...              scale_factor=2.0))
The island is one of the colony?" asked the sailor, "there is not to be
able to come to the shores of the Pacific."
"Yes," replied the engineer, "and if it is not the position of the
forest, and the marshy way have been said, the dog was not first on the
shore, and
found themselves to the corral.
```

```
The settlers had the sailor was still from the surface of the sea, they
were not received for the sea. The shore was to be able to inspect the
windows of Granite House.
The sailor turned the sailor was the hor
```

- $\alpha = 0.5 \rightarrow$ more randomness:

```
>>> torch.manual_seed(1)
>>> print(sample(model, starting_str='The island',
...              scale_factor=0.5))
The island
deep incomele.
Manyl's', House, won's calcon-sglenderlessly," everful ineriorouins.,
pyra" into
truth. Sometinivabes, iskumar gave-zen."

Bleshed but what cotch quadrap which little cedass
fell oprely
by-andonem. Peditivall--"i dove Gurgeon. What resolt-eartnated to him
ran trail.

Withinhe)tiny turns returned, after owner plan bushelsion lairs; they
were
know? Whalerin branch I
pites, Dougg!-iteun," returnwe aid masses atong thoughts! Dak,
Hem-arches yone, Veay wantzer? Woblding,
Herbert, omep
```

The results show that scaling the logits with $\alpha = 0.5$ (increasing the temperature) generates more random text. There is a trade-off between the novelty of the generated text and its correctness.

In this section, we worked with character-level text generation, which is a sequence-to-sequence (seq2seq) modeling task. While this example may not be very useful by itself, it is easy to think of several useful applications for these types of models; for example, a similar RNN model can be trained as a chatbot to assist users with simple queries.

Summary

In this chapter, you first learned about the properties of sequences that make them different from other types of data, such as structured data or images. We then covered the foundations of RNNs for sequence modeling. You learned how a basic RNN model works and discussed its limitations with regard to capturing long-term dependencies in sequence data. Next, we covered LSTM cells, which consist of a gating mechanism to reduce the effect of exploding and vanishing gradient problems, which are common in basic RNN models.

After discussing the main concepts behind RNNs, we implemented several RNN models with different recurrent layers using PyTorch. In particular, we implemented an RNN model for sentiment analysis, as well as an RNN model for generating text.

In the next chapter, we will see how we can augment an RNN with an attention mechanism, which helps it with modeling long-range dependencies in translation tasks. Then, we will introduce a new deep learning architecture called *transformer*, which has recently been used to further push the state of the art in the natural language processing domain.

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:
`https://packt.link/MLwPyTorch`