CHAPTER 5
Autoregressive Models

Chapter Goals

In this chapter you will:

- Learn why autoregressive models are well suited to generating sequential data such as text.
- Learn how to process and tokenize text data.
- Learn about the architectural design of recurrent neural networks (RNNs).
- Build and train a long short-term memory network (LSTM) from scratch using Keras.
- Use the LSTM to generate new text.
- Learn about other variations of RNNs, including gated recurrent units (GRUs) and bidirectional cells.
- Understand how image data can be treated as a sequence of pixels.
- Learn about the architectural design of a PixelCNN.
- Build a PixelCNN from scratch using Keras.
- Use the PixelCNN to generate images.

So far, we have explored two different families of generative models that have both involved latent variables—variational autoencoders (VAEs) and generative adversarial networks (GANs). In both cases, a new variable is introduced with a distribution that is easy to sample from and the model learns how to *decode* this variable back into the original domain.

We will now turn our attention to *autoregressive models*—a family of models that simplify the generative modeling problem by treating it as a sequential process. Autoregressive models condition predictions on previous values in the sequence, rather than on a latent random variable. Therefore, they attempt to explicitly model the data-generating distribution rather than an approximation of it (as in the case of VAEs).

In this chapter we shall explore two different autoregressive models: long short-term memory networks and PixelCNN. We will apply the LSTM to text data and the PixelCNN to image data. We will cover another highly successful autoregressive model, the Transformer, in detail in Chapter 9.

Introduction

To understand how an LSTM works, we will first pay a visit to a strange prison, where the inmates have formed a literary society…

The Literary Society for Troublesome Miscreants

Edward Sopp hated his job as a prison warden. He spent his days watching over the prisoners and had no time to follow his true passion of writing short stories. He was running low on inspiration and needed to find a way to generate new content.

One day, he came up with a brilliant idea that would allow him to produce new works of fiction in his style, while also keeping the inmates occupied—he would get the inmates to collectively write the stories for him! He branded the new society the Literary Society for Troublesome Miscreants, or LSTM (Figure 5-1).

Figure 5-1. A large cell of prisoners reading books (created with Midjourney)

The prison is particularly strange because it only consists of one large cell, containing 256 prisoners. Each prisoner has an opinion on how Edward's current story should continue. Every day, Edward posts the latest word from his novel into the cell, and it is the job of the inmates to individually update their opinions on the current state of the story, based on the new word and the opinions of the inmates from the previous day.

Each prisoner uses a specific thought process to update their own opinion, which involves balancing information from the new incoming word and other prisoners' opinions with their own prior beliefs. First, they decide how much of yesterday's opinion they wish to forget, taking into account the information from the new word and the opinions of other prisoners in the cell. They also use this information to form new thoughts and decide to what extent they want to mix these into the old beliefs that they have chosen to carry forward from the previous day. This then forms the prisoner's new opinion for the day.

However, the prisoners are secretive and don't always tell their fellow inmates all of their opinions. They each also use the latest chosen word and the opinions of the other inmates to decide how much of their opinion they wish to disclose.

When Edward wants the cell to generate the next word in the sequence, the prisoners each tell their disclosable opinions to the guard at the door, who combines this information to ultimately decide on the next word to be appended to the end of the novel. This new word is then fed back into the cell, and the process continues until the full story is completed.

To train the inmates and the guard, Edward feeds short sequences of words that he has written previously into the cell and monitors whether the inmates' chosen next word is correct. He updates them on their accuracy, and gradually they begin to learn how to write stories in his own unique style.

After many iterations of this process, Edward finds that the system has become quite accomplished at generating realistic-looking text. Satisfied with the results, he publishes a collection of the generated tales in his new book, entitled *E. Sopp's Fables*.

The story of Mr. Sopp and his crowdsourced fables is an analogy for one of the most notorious autoregressive techniques for sequential data such as text: the long short-term memory network.

Long Short-Term Memory Network (LSTM)

An LSTM is a particular type of recurrent neural network (RNN). RNNs contain a recurrent layer (or *cell*) that is able to handle sequential data by making its own output at a particular timestep form part of the input to the next timestep.

When RNNs were first introduced, recurrent layers were very simple and consisted solely of a tanh operator that ensured that the information passed between timesteps was scaled between –1 and 1. However, this approach was shown to suffer from the vanishing gradient problem and didn't scale well to long sequences of data.

LSTM cells were first introduced in 1997 in a paper by Sepp Hochreiter and Jürgen Schmidhuber.[1] In the paper, the authors describe how LSTMs do not suffer from the same vanishing gradient problem experienced by vanilla RNNs and can be trained on sequences that are hundreds of timesteps long. Since then, the LSTM architecture has been adapted and improved, and variations such as gated recurrent units (discussed later in this chapter) are now widely utilized and available as layers in Keras.

LSTMs have been applied to a wide range of problems involving sequential data, including time series forecasting, sentiment analysis, and audio classification. In this chapter we will be using LSTMs to tackle the challenge of text generation.

Running the Code for This Example

The code for this example can be found in the Jupyter notebook located at *notebooks/05_autoregressive/01_lstm/lstm.ipynb* in the book repository.

The Recipes Dataset

We'll be using the Epicurious Recipes dataset that is available through Kaggle. This is a set of over 20,000 recipes, with accompanying metadata such as nutritional information and ingredient lists.

You can download the dataset by running the Kaggle dataset downloader script in the book repository, as shown in Example 5-1. This will save the recipes and accompanying metadata locally to the */data* folder.

Example 5-1. Downloading the Epicurious Recipe dataset

```
bash scripts/download_kaggle_data.sh hugodarwood epirecipes
```

Example 5-2 shows how the data can be loaded and filtered so that only recipes with a title and a description remain. An example of a recipe text string is given in Example 5-3.

Example 5-2. Loading the data

```
with open('/app/data/epirecipes/full_format_recipes.json') as json_data:
    recipe_data = json.load(json_data)

filtered_data = [
```

```
    'Recipe for ' + x['title']+ ' | ' + ' '.join(x['directions'])
    for x in recipe_data
    if 'title' in x
    and x['title'] is not None
    and 'directions' in x
    and x['directions'] is not None
]
```

Example 5-3. A text string from the Recipes dataset

```
Recipe for Ham Persillade with Mustard Potato Salad and Mashed Peas  | Chop enough
parsley leaves to measure 1 tablespoon; reserve. Chop remaining leaves and stems
and simmer with broth and garlic in a small saucepan, covered, 5 minutes.
Meanwhile, sprinkle gelatin over water in a medium bowl and let soften 1 minute.
Strain broth through a fine-mesh sieve into bowl with gelatin and stir to dissolve.
Season with salt and pepper. Set bowl in an ice bath and cool to room temperature,
stirring. Toss ham with reserved parsley and divide among jars. Pour gelatin on top
and chill until set, at least 1 hour. Whisk together mayonnaise, mustard, vinegar,
1/4 teaspoon salt, and 1/4 teaspoon pepper in a large bowl. Stir in celery,
cornichons, and potatoes. Pulse peas with marjoram, oil, 1/2 teaspoon pepper, and
1/4 teaspoon salt in a food processor to a coarse mash. Layer peas, then potato
salad, over ham.
```

Before taking a look at how to build an LSTM network in Keras, we must first take a quick detour to understand the structure of text data and how it is different from the image data that we have seen so far in this book.

Working with Text Data

There are several key differences between text and image data that mean that many of the methods that work well for image data are not so readily applicable to text data. In particular:

- Text data is composed of discrete chunks (either characters or words), whereas pixels in an image are points in a continuous color spectrum. We can easily make a green pixel more blue, but it is not obvious how we should go about making the word *cat* more like the word *dog*, for example. This means we can easily apply backpropagation to image data, as we can calculate the gradient of our loss function with respect to individual pixels to establish the direction in which pixel colors should be changed to minimize the loss. With discrete text data, we can't obviously apply backpropagation in the same way, so we need to find a way around this problem.
- Text data has a time dimension but no spatial dimension, whereas image data has two spatial dimensions but no time dimension. The order of words is highly important in text data and words wouldn't make sense in reverse, whereas images can usually be flipped without affecting the content. Furthermore, there are often

long-term sequential dependencies between words that need to be captured by the model: for example, the answer to a question or carrying forward the context of a pronoun. With image data, all pixels can be processed simultaneously.

- Text data is highly sensitive to small changes in the individual units (words or characters). Image data is generally less sensitive to changes in individual pixel units—a picture of a house would still be recognizable as a house even if some pixels were altered—but with text data, changing even a few words can drastically alter the meaning of the passage, or make it nonsensical. This makes it very difficult to train a model to generate coherent text, as every word is vital to the overall meaning of the passage.
- Text data has a rules-based grammatical structure, whereas image data doesn't follow set rules about how the pixel values should be assigned. For example, it wouldn't make grammatical sense in any context to write "The cat sat on the having." There are also semantic rules that are extremely difficult to model; it wouldn't make sense to say "I am in the beach," even though grammatically, there is nothing wrong with this statement.

Advances in Text-Based Generative Deep Learning

Until recently, most of the most sophisticated generative deep learning models have focused on image data, because many of the challenges presented in the preceding list were beyond the reach of even the most advanced techniques. However, in the last five years astonishing progress has been made in the field of text-based generative deep learning, thanks to the introduction of the Transformer model architecture, which we will explore in Chapter 9.

With these points in mind, let's now take a look at the steps we need to take in order to get the text data into the right shape to train an LSTM network.

Tokenization

The first step is to clean up and tokenize the text. *Tokenization* is the process of splitting the text up into individual units, such as words or characters.

How you tokenize your text will depend on what you are trying to achieve with your text generation model. There are pros and cons to using both word and character tokens, and your choice will affect how you need to clean the text prior to modeling and the output from your model.

If you use word tokens:

- All text can be converted to lowercase, to ensure capitalized words at the start of sentences are tokenized the same way as the same words appearing in the middle of a sentence. In some cases, however, this may not be desirable; for example, some proper nouns, such as names or places, may benefit from remaining capitalized so that they are tokenized independently.
- The text *vocabulary* (the set of distinct words in the training set) may be very large, with some words appearing very sparsely or perhaps only once. It may be wise to replace sparse words with a token for *unknown word*, rather than including them as separate tokens, to reduce the number of weights the neural network needs to learn.
- Words can be *stemmed*, meaning that they are reduced to their simplest form, so that different tenses of a verb remained tokenized together. For example, *browse*, *browsing*, *browses*, and *browsed* would all be stemmed to *brows*.
- You will need to either tokenize the punctuation, or remove it altogether.
- Using word tokenization means that the model will never be able to predict words outside of the training vocabulary.

If you use character tokens:

- The model may generate sequences of characters that form new words outside of the training vocabulary—this may be desirable in some contexts, but not in others.
- Capital letters can either be converted to their lowercase counterparts, or remain as separate tokens.
- The vocabulary is usually much smaller when using character tokenization. This is beneficial for model training speed as there are fewer weights to learn in the final output layer.

For this example, we'll use lowercase word tokenization, without word stemming. We'll also tokenize punctuation marks, as we would like the model to predict when it should end sentences or use commas, for example.

The code in Example 5-4 cleans and tokenizes the text.

Example 5-4. Tokenization

```
def pad_punctuation(s):
    s = re.sub(f"([{string.punctuation}])", r' \1 ', s)
    s = re.sub(' +', ' ', s)
    return s
```

```
text_data = [pad_punctuation(x) for x in filtered_data] ❶

text_ds = tf.data.Dataset.from_tensor_slices(text_data).batch(32).shuffle(1000) ❷

vectorize_layer = layers.TextVectorization( ❸
    standardize = 'lower',
    max_tokens = 10000,
    output_mode = "int",
    output_sequence_length = 200 + 1,
)

vectorize_layer.adapt(text_ds) ❹
vocab = vectorize_layer.get_vocabulary() ❺
```

❶ Pad the punctuation marks, to treat them as separate words.

❷ Convert to a TensorFlow Dataset.

❸ Create a Keras `TextVectorization` layer to convert text to lowercase, give the most prevalent 10,000 words a corresponding integer token, and trim or pad the sequence to 201 tokens long.

❹ Apply the `TextVectorization` layer to the training data.

❺ The `vocab` variable stores a list of the word tokens.

An example of a recipe after tokenization is shown in Example 5-5. The sequence length that we use to train the model is a parameter of the training process. In this example we choose to use a sequence length of 200, so we pad or clip the recipe to one more than this length, to allow us to create the target variable (more on this in the next section). To achieve this desired length, the end of the vector is padded with zeros.

Stop Tokens

The `0` token is known as a the *stop token*, signifying that the text string has come to an end.

Example 5-5. The recipe from Example 5-3 tokenized

```
[  26   16  557    1    8  298  335  189    4 1054  494   27  332  228
  235  262    5  594   11  133   22  311    2  332   45  262    4  671
    4   70    8  171    4   81    6    9   65   80    3  121    3   59
   12    2  299    3   88  650   20   39    6    9   29   21    4   67
  529   11  164    2  320  171  102    9  374   13  643  306   25   21
    8  650    4   42    5  931    2   63    8   24    4   33    2  114
   21    6  178  181 1245    4   60    5  140  112    3   48    2  117
```

```
  557    8  285  235    4  200  292  980    2  107  650   28   72    4
  108   10  114    3   57  204   11  172    2   73  110  482    3  298
    3  190    3   11   23   32  142   24    3    4   11   23   32  142
   33    6    9   30   21    2   42    6  353    3 3224    3    4  150
    2  437  494    8 1281    3   37    3   11   23   15  142   33    3
    4   11   23   32  142   24    6    9  291  188    5    9  412  572
    2  230  494    3   46  335  189    3   20  557    2    0    0    0
    0    0    0    0    0]
```

In Example 5-6, we can see a subset of the list of tokens mapped to their respective indices. The layer reserves the `0` token for padding (i.e., it is the stop token) and the `1` token for unknown words that fall outside the top 10,000 words (e.g., persillade). The other words are assigned tokens in order of frequency. The number of words to include in the vocabulary is also a parameter of the training process. The more words included, the fewer *unknown* tokens you will see in the text; however, your model will need to be larger to accommodate the larger vocabulary size.

Example 5-6. The vocabulary of the `TextVectorization` layer

```
0:
1: [UNK]
2: .
3: ,
4: and
5: to
6: in
7: the
8: with
9: a
```

Creating the Training Set

Our LSTM will be trained to predict the next word in a sequence, given a sequence of words preceding this point. For example, we could feed the model the tokens for *grilled chicken with boiled* and would expect the model to output a suitable next word (e.g., *potatoes*, rather than *bananas*).

We can therefore simply shift the entire sequence by one token in order to create our target variable.

The dataset generation step can be achieved with the code in Example 5-7.

Example 5-7. Creating the training dataset

```
def prepare_inputs(text):
    text = tf.expand_dims(text, -1)
    tokenized_sentences = vectorize_layer(text)
    x = tokenized_sentences[:, :-1]
```

```
    y = tokenized_sentences[:, 1:]
    return x, y

train_ds = text_ds.map(prepare_inputs) ❶
```

❶ Create the training set consisting of recipe tokens (the input) and the same vector shifted by one token (the target).

The LSTM Architecture

The architecture of the overall LSTM model is shown in Table 5-1. The input to the model is a sequence of integer tokens and the output is the probability of each word in the 10,000-word vocabulary appearing next in the sequence. To understand how this works in detail, we need to introduce two new layer types, `Embedding` and `LSTM`.

Table 5-1. Model summary of the LSTM

Layer (type)	Output shape	Param #
InputLayer	(None, None)	0
Embedding	(None, None, 100)	1,000,000
LSTM	(None, None, 128)	117,248
Dense	(None, None, 10000)	1,290,000

Total params	2,407,248
Trainable params	2,407,248
Non-trainable params	0

The Input Layer of the LSTM

Notice that the `Input` layer does not need us to specify the sequence length in advance. Both the batch size and the sequence length are flexible (hence the `(None, None)` shape). This is because all downstream layers are agnostic to the length of the sequence being passed through.

The Embedding Layer

An *embedding layer* is essentially a lookup table that converts each integer token into a vector of length `embedding_size`, as shown in Figure 5-2. The lookup vectors are learned by the model as *weights*. Therefore, the number of weights learned by this layer is equal to the size of the vocabulary multiplied by the dimension of the embedding vector (i.e., 10,000 × 100 = 1,000,000).

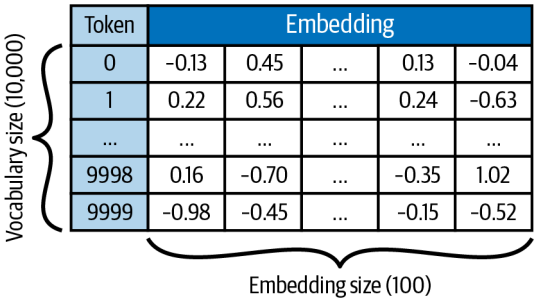

Figure 5-2. An embedding layer is a lookup table for each integer token

We embed each integer token into a continuous vector because it enables the model to learn a representation for each word that is able to be updated through backpropagation. We could also just one-hot encode each input token, but using an embedding layer is preferred because it makes the embedding itself trainable, thus giving the model more flexibility in deciding how to embed each token to improve its performance.

Therefore, the `Input` layer passes a tensor of integer sequences of shape `[batch_size, seq_length]` to the `Embedding` layer, which outputs a tensor of shape `[batch_size, seq_length, embedding_size]`. This is then passed on to the `LSTM` layer (Figure 5-3).

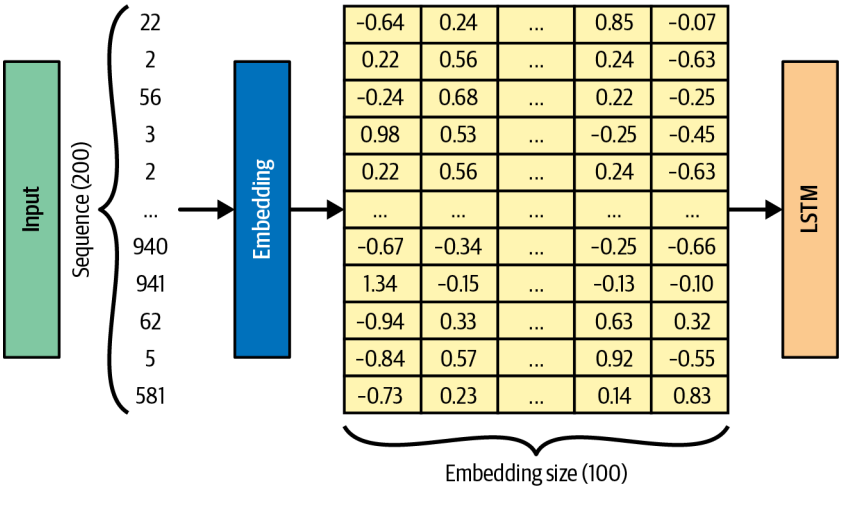

Figure 5-3. A single sequence as it flows through an embedding layer

The LSTM Layer

To understand the LSTM layer, we must first look at how a general recurrent layer works.

A recurrent layer has the special property of being able to process sequential input data $x_1, \cdots, x_n$. It consists of a cell that updates its *hidden state*, h_t, as each element of the sequence x_t is passed through it, one timestep at a time.

The hidden state is a vector with length equal to the number of *units* in the cell—it can be thought of as the cell's current understanding of the sequence. At timestep t, the cell uses the previous value of the hidden state, h_{t-1}, together with the data from the current timestep x_t to produce an updated hidden state vector, h_t. This recurrent process continues until the end of the sequence. Once the sequence is finished, the layer outputs the final hidden state of the cell, h_n, which is then passed on to the next layer of the network. This process is shown in Figure 5-4.

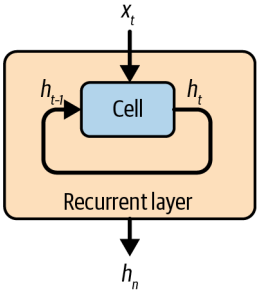

Figure 5-4. A simple diagram of a recurrent layer

To explain this in more detail, let's unroll the process so that we can see exactly how a single sequence is fed through the layer (Figure 5-5).

Cell Weights

It's important to remember that all of the cells in this diagram share the same weights (as they are really the same cell). There is no difference between this diagram and Figure 5-4; it's just a different way of drawing the mechanics of a recurrent layer.

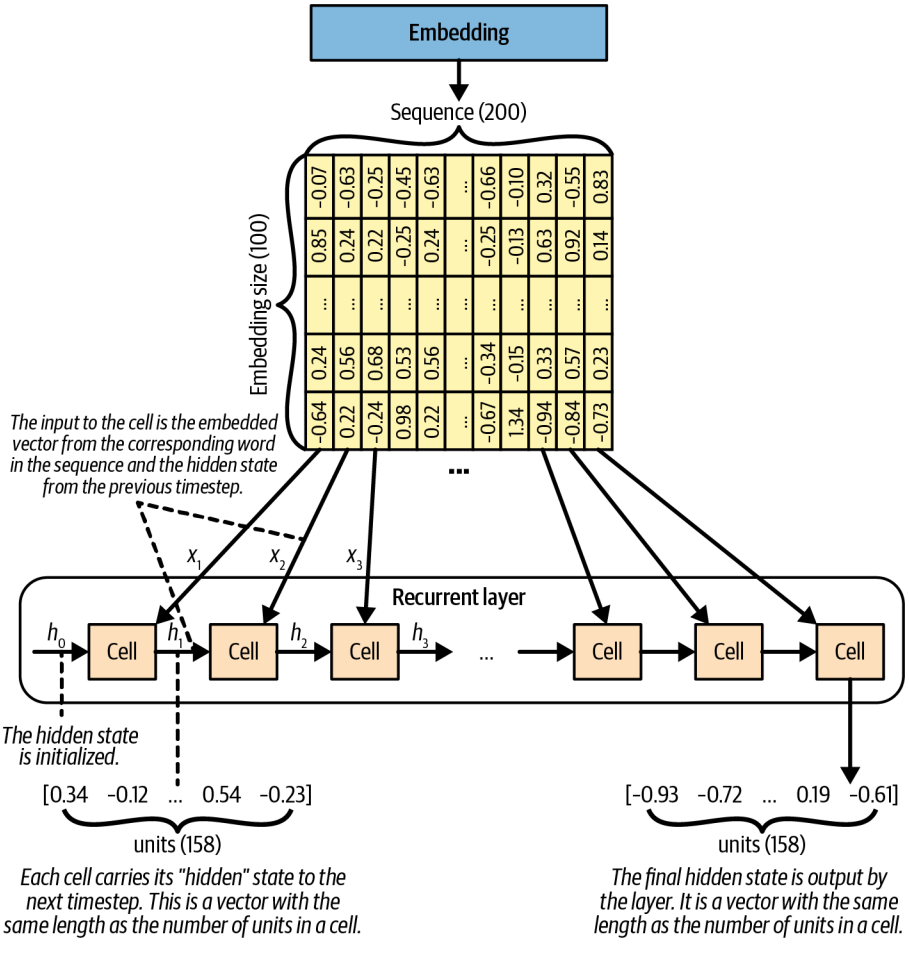

Figure 5-5. How a single sequence flows through a recurrent layer

Here, we represent the recurrent process by drawing a copy of the cell at each time-step and show how the hidden state is constantly being updated as it flows through the cells. We can clearly see how the previous hidden state is blended with the current sequential data point (i.e., the current embedded word vector) to produce the next hidden state. The output from the layer is the final hidden state of the cell, after each word in the input sequence has been processed.

The fact that the output from the cell is called a *hidden* state is an unfortunate naming convention—it's not really hidden, and you shouldn't think of it as such. Indeed, the last hidden state is the overall output from the layer, and we will be making use of the fact that we can access the hidden state at each individual timestep later in this chapter.

The LSTM Cell

Now that we have seen how a generic recurrent layer works, let's take a look inside an individual LSTM cell.

The job of the LSTM cell is to output a new hidden state, h_t, given its previous hidden state, h_{t-1}, and the current word embedding, x_t. To recap, the length of h_t is equal to the number of units in the LSTM. This is a parameter that is set when you define the layer and has nothing to do with the length of the sequence.

Make sure you do not confuse the term *cell* with *unit*. There is one cell in an LSTM layer that is defined by the number of units it contains, in the same way that the prisoner cell from our earlier story contained many prisoners. We often draw a recurrent layer as a chain of cells unrolled, as it helps to visualize how the hidden state is updated at each timestep.

An LSTM cell maintains a cell state, C_t, which can be thought of as the cell's internal beliefs about the current status of the sequence. This is distinct from the hidden state, h_t, which is ultimately output by the cell after the final timestep. The cell state is the same length as the hidden state (the number of units in the cell).

Let's look more closely at a single cell and how the hidden state is updated (Figure 5-6).

The hidden state is updated in six steps:

1. The hidden state of the previous timestep, h_{t-1}, and the current word embedding, x_t, are concatenated and passed through the *forget* gate. This gate is simply a dense layer with weights matrix W_f, bias b_f, and a sigmoid activation function. The resulting vector, f_t, has length equal to the number of units in the cell and contains values between 0 and 1 that determine how much of the previous cell state, C_{t-1}, should be retained.

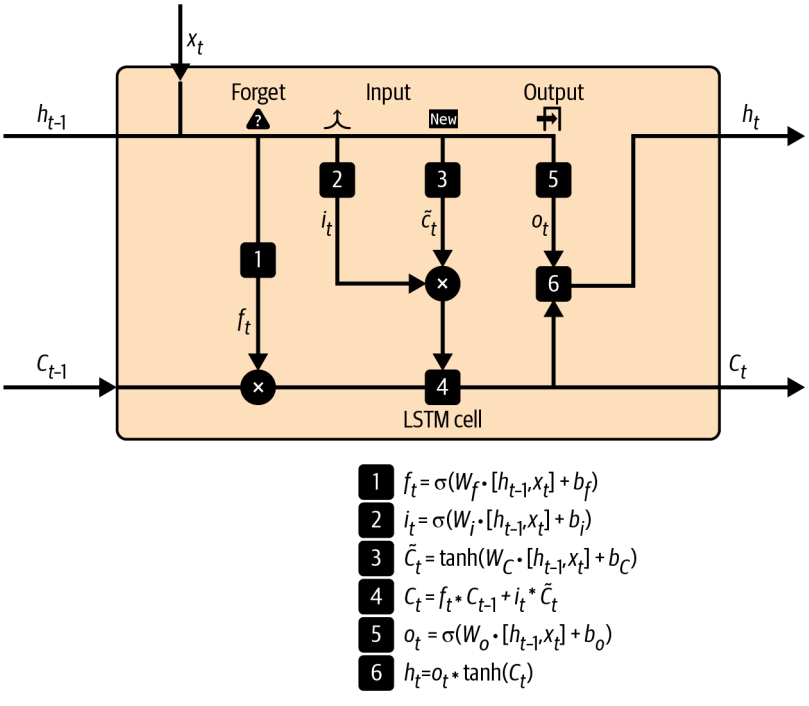

Figure 5-6. An LSTM cell

2. The concatenated vector is also passed through an *input* gate that, like the forget gate, is a dense layer with weights matrix W_i, bias b_i, and a sigmoid activation function. The output from this gate, i_t, has length equal to the number of units in the cell and contains values between 0 and 1 that determine how much new information will be added to the previous cell state, C_{t-1}.
3. The concatenated vector is passed through a dense layer with weights matrix W_C, bias b_C, and a tanh activation function to generate a vector $\tilde{C}_t$ that contains the new information that the cell wants to consider keeping. It also has length equal to the number of units in the cell and contains values between –1 and 1.
4. f_t and C_{t-1} are multiplied element-wise and added to the element-wise multiplication of i_t and $\tilde{C}_t$. This represents forgetting parts of the previous cell state and then adding new relevant information to produce the updated cell state, C_t.
5. The concatenated vector is passed through an *output* gate: a dense layer with weights matrix W_o, bias b_o, and a sigmoid activation. The resulting vector, o_t, has length equal to the number of units in the cell and stores values between 0 and 1 that determine how much of the updated cell state, C_t, to output from the cell.

6. o_t is multiplied element-wise with the updated cell state, C_t, after a tanh activation has been applied to produce the new hidden state, h_t.

The Keras LSTM Layer

All of this complexity is wrapped up within the `LSTM` layer type in Keras, so you don't have to worry about implementing it yourself!

Training the LSTM

The code to build, compile, and train the LSTM is given in Example 5-8.

Example 5-8. Building, compiling, and training the LSTM

```
inputs = layers.Input(shape=(None,), dtype="int32") ❶
x = layers.Embedding(10000, 100)(inputs) ❷
x = layers.LSTM(128, return_sequences=True)(x) ❸
outputs = layers.Dense(10000, activation = 'softmax')(x) ❹
lstm = models.Model(inputs, outputs) ❺

loss_fn = losses.SparseCategoricalCrossentropy()
lstm.compile("adam", loss_fn) ❻
lstm.fit(train_ds, epochs=25) ❼
```

❶ The `Input` layer does not need us to specify the sequence length in advance (it can be flexible), so we use `None` as a placeholder.

❷ The `Embedding` layer requires two parameters, the size of the vocabulary (10,000 tokens) and the dimensionality of the embedding vector (100).

❸ The LSTM layers require us to specify the dimensionality of the hidden vector (128). We also choose to return the full sequence of hidden states, rather than just the hidden state at the final timestep.

❹ The `Dense` layer transforms the hidden states at each timestep into a vector of probabilities for the next token.

❺ The overall `Model` predicts the next token, given an input sequence of tokens. It does this for each token in the sequence.

❻ The model is compiled with `SparseCategoricalCrossentropy` loss—this is the same as categorical cross-entropy, but is used when the labels are integers rather than one-hot encoded vectors.

❼ The model is fit to the training dataset.

In Figure 5-7 you can see the first few epochs of the LSTM training process—notice how the example output becomes more comprehensible as the loss metric falls. Figure 5-8 shows the cross-entropy loss metric falling throughout the training process.

```
Epoch 1/25
628/629 [============================>.] - ETA: 0s - loss: 4.4536
generated text:
recipe for mold salad are high 8 pickled to fold cook the dish into and warm in baking reduced but halves beans
and cut

629/629 [==============================] - 29s 43ms/step - loss: 4.4527
Epoch 2/25
628/629 [============================>.] - ETA: 0s - loss: 3.2339
generated text:
recipe for racks - up—don with herb fizz | serve checking thighs onto sanding butter and baking surface in a hea
vy heavy large saucepan over blender ; stand overnight . [UNK] over moderate heat until very blended , garlic ,
about 8 minutes . cook airtight until cooked are soft seeds , about 1 45 minutes . sugar , until s is brown , 5
to sliced , parmesan , until browned and add extract . wooden crumb to outside of out sheets . flatten and prehe
ated return to the paste . add in pecans oval and let transfer day .

629/629 [==============================] - 30s 48ms/step - loss: 3.2336
Epoch 3/25
629/629 [==============================] - ETA: 0s - loss: 2.6229
generated text:
recipe for grilled chicken | preheat oven to 400°f . cook in large 8 - caramel grinder or until desired are firm
, about 6 minutes

629/629 [==============================] - 27s 42ms/step - loss: 2.6229
Epoch 4/25
629/629 [==============================] - ETA: 0s - loss: 2.3426
generated text:
recipe for pizza salad with sweet red pepper and star fruit | combine all ingredients except lowest ingredients
in a large skillet . working with batches and deglaze , cook until just cooked through , about 1 minute . meanwh
ile , boil potatoes and paprika in a little oil over medium - high heat , stirring it just until crisp , about 3
minutes . stir in bell pepper , onion and cooked paste and jalapeño until clams well after most - reggiano , abo
ut 5 minutes . transfer warm 2 tablespoons flesh of eggplants to medium bowl . serve .
```

Figure 5-7. The first few epochs of the LSTM training process

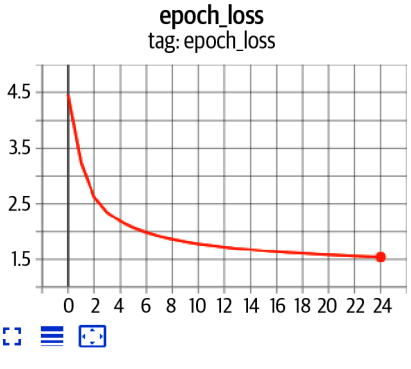

Figure 5-8. The cross-entropy loss metric of the LSTM training process by epoch

Analysis of the LSTM

Now that we have compiled and trained the LSTM, we can start to use it to generate long strings of text by applying the following process:

1. Feed the network with an existing sequence of words and ask it to predict the following word.
2. Append this word to the existing sequence and repeat.

The network will output a set of probabilities for each word that we can sample from. Therefore, we can make the text generation stochastic, rather than deterministic. Moreover, we can introduce a *temperature* parameter to the sampling process to indicate how deterministic we would like the process to be.

The Temperature Parameter

A temperature close to 0 makes the sampling more deterministic (i.e., the word with the highest probability is very likely to be chosen), whereas a temperature of 1 means each word is chosen with the probability output by the model.

This is achieved with the code in Example 5-9, which creates a callback function that can be used to generate text at the end of each training epoch.

Example 5-9. The `TextGenerator` callback function

```
class TextGenerator(callbacks.Callback):
    def __init__(self, index_to_word, top_k=10):
        self.index_to_word = index_to_word
        self.word_to_index = {
            word: index for index, word in enumerate(index_to_word)
        } ❶

    def sample_from(self, probs, temperature): ❷
        probs = probs ** (1 / temperature)
        probs = probs / np.sum(probs)
        return np.random.choice(len(probs), p=probs), probs

    def generate(self, start_prompt, max_tokens, temperature):
        start_tokens = [
            self.word_to_index.get(x, 1) for x in start_prompt.split()
        ] ❸
        sample_token = None
        info = []
        while len(start_tokens) < max_tokens and sample_token != 0: ❹
            x = np.array([start_tokens])
            y = self.model.predict(x) ❺
```

```
            sample_token, probs = self.sample_from(y[0][-1], temperature) ❻
            info.append({'prompt': start_prompt , 'word_probs': probs})
            start_tokens.append(sample_token) ❼
            start_prompt = start_prompt + ' ' + self.index_to_word[sample_token]
        print(f"\ngenerated text:\n{start_prompt}\n")
        return info

    def on_epoch_end(self, epoch, logs=None):
        self.generate("recipe for", max_tokens = 100, temperature = 1.0)
```

❶ Create an inverse vocabulary mapping (from word to token).

❷ This function updates the probabilities with a `temperature` scaling factor.

❸ The start prompt is a string of words that you would like to give the model to start the generation process (for example, *recipe for*). The words are first converted to a list of tokens.

❹ The sequence is generated until it is `max_tokens` long or a stop token (0) is produced.

❺ The model outputs the probabilities of each word being next in the sequence.

❻ The probabilities are passed through the sampler to output the next word, parameterized by `temperature`.

❼ We append the new word to the prompt text, ready for the next iteration of the generative process.

Let's take a look at this in action, at two different temperature values (Figure 5-9).

```
temperature = 1.0

generated text:
recipe for sour japanese potatoes julienne | in a bowl stir together the yeast mixture with the milk and the
peanut butter crumbs , the sour cream , and the butter mixture with a fork , gently fold in the prunes gently
or until incorporated . lightly stir the oil and yeast until it just holds soft peaks , but not runny , on bo
ttom of a 7 - sided sheet of aluminum foil , top it with a round , and a pinch of each brownies into a goblet
, or with the baking dish . serve each with sorbet

temperature = 0.2

generated text:
recipe for grilled chicken with mustard - herb sauce | combine first 6 ingredients in medium bowl . add chick
en to pot . add chicken and turn to coat . cover and refrigerate at least 1 hour and up to 1 day . preheat ov
en to 450°f . place turkey on rock in roasting pan . roast until thermometer inserted into thickest part of t
high registers 175°f , about 1 hour longer . transfer to rack in center of oven and preheat to 450°f . brush
chicken with oil . sprinkle with salt and pepper . roast until thermometer inserted into
```

Figure 5-9. Generated outputs at `temperature = 1.0` and `temperature = 0.2`

There are a few things to note about these two passages. First, both are stylistically similar to a recipe from the original training set. They both open with a recipe title and contain generally grammatically correct constructions. The difference is that the generated text with a temperature of 1.0 is more adventurous and therefore less accurate than the example with a temperature of 0.2. Generating multiple samples with a temperature of 1.0 will therefore lead to more variety, as the model is sampling from a probability distribution with greater variance.

To demonstrate this, Figure 5-10 shows the top five tokens with the highest probabilities for a range of prompts, for both temperature values.

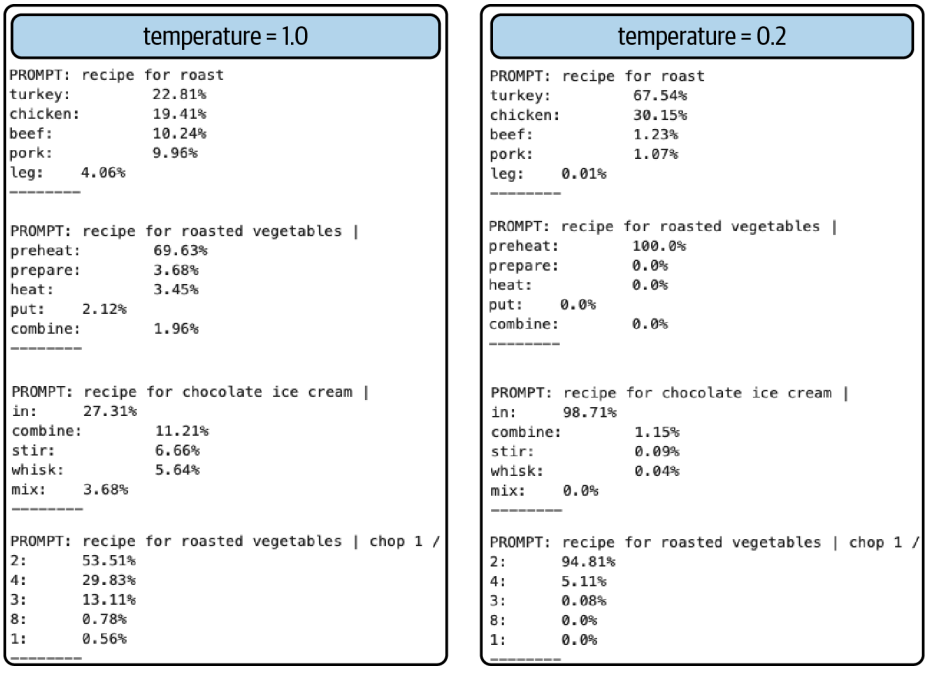

Figure 5-10. Distribution of word probabilities following various sequences, for temperature values of 1.0 and 0.2

The model is able to generate a suitable distribution for the next most likely word across a range of contexts. For example, even though the model was never told about parts of speech such as nouns, verbs, or numbers, it is generally able to separate words into these classes and use them in a way that is grammatically correct.

Moreover, the model is able to select an appropriate verb to begin the recipe instructions, depending on the preceding title. For roasted vegetables, it selects `preheat`, `prepare`, `heat`, `put`, or `combine` as the most likely possibilities, whereas for ice cream it selects `in`, `combine`, `stir`, `whisk`, and `mix`. This shows that the model has some contextual understanding of the differences between recipes depending on their ingredients.

Notice also how the probabilities for the `temperature = 0.2` examples are much more heavily weighted toward the first choice token. This is the reason why there is generally less variety in generations when the temperature is lower.

While our basic LSTM model is doing a great job at generating realistic text, it is clear that it still struggles to grasp some of the semantic meaning of the words that it is generating. It introduces ingredients that are not likely to work well together (for example, sour Japanese potatoes, pecan crumbs, and sorbet)! In some cases, this may be desirable—say, if we want our LSTM to generate interesting and unique patterns of words—but in other cases, we will need our model to have a deeper understanding of the ways in which words can be grouped together and a longer memory of ideas introduced earlier in the text.

In the next section, we'll explore some of the ways that we can improve our basic LSTM network. In Chapter 9, we'll take a look at a new kind of autoregressive model, the Transformer, which takes language modeling to the next level.

Recurrent Neural Network (RNN) Extensions

The model in the preceding section is a simple example of how an LSTM can be trained to learn how to generate text in a given style. In this section we will explore several extensions to this idea.

Stacked Recurrent Networks

The network we just looked at contained a single LSTM layer, but we can also train networks with stacked LSTM layers, so that deeper features can be learned from the text.

To achieve this, we simply introduce another LSTM layer after the first. The second LSTM layer can then use the hidden states from the first layer as its input data. This is shown in Figure 5-11, and the overall model architecture is shown in Table 5-2.

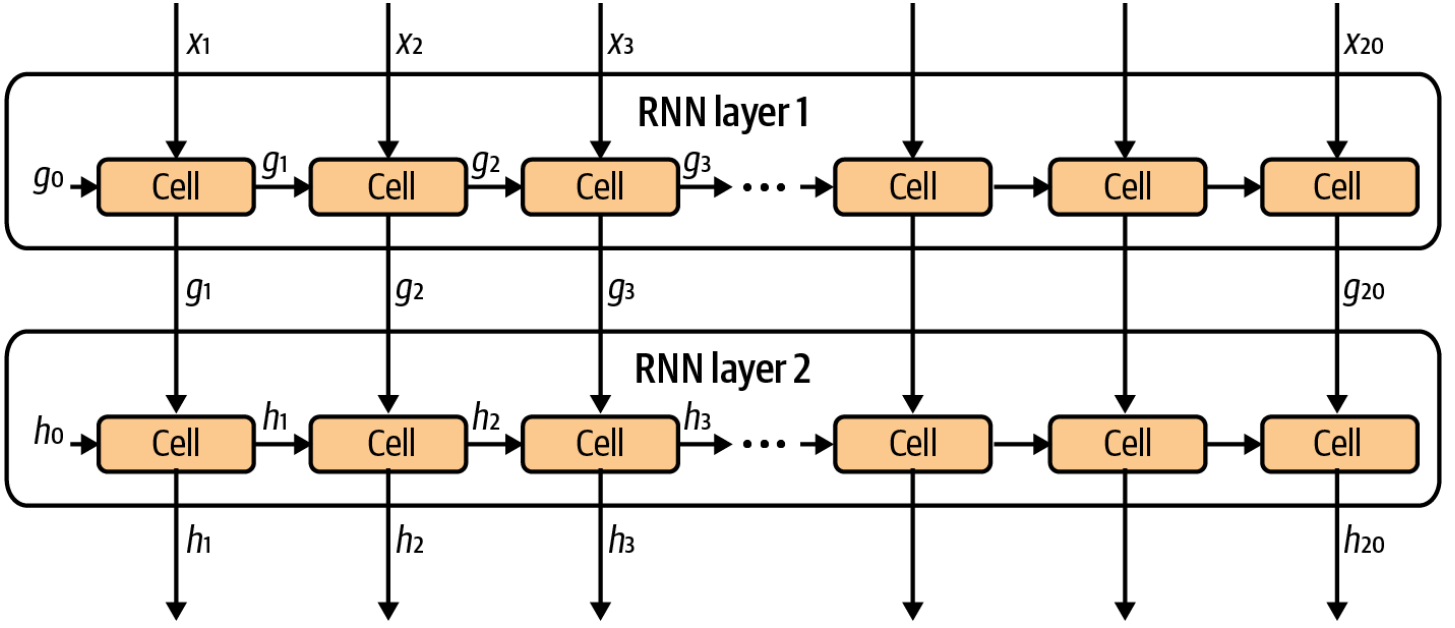

Figure 5-11. Diagram of a multilayer RNN: g_t denotes hidden states of the first layer and h_t denotes hidden states of the second layer

Table 5-2. Model summary of the stacked LSTM

Layer (type)	Output shape	Param #
InputLayer	(None, None)	0
Embedding	(None, None, 100)	1,000,000
LSTM	(None, None, 128)	117,248
LSTM	(None, None, 128)	131,584
Dense	(None, None, 10000)	1,290,000

Total params	2,538,832
Trainable params	2,538,832
Non-trainable params	0

The code to build the stacked LSTM is given in Example 5-10.

Example 5-10. Building a stacked LSTM

```
text_in = layers.Input(shape = (None,))
embedding = layers.Embedding(total_words, embedding_size)(text_in)
x = layers.LSTM(n_units, return_sequences = True)(x)
x = layers.LSTM(n_units, return_sequences = True)(x)
probabilites = layers.Dense(total_words, activation = 'softmax')(x)
model = models.Model(text_in, probabilites)
```

Gated Recurrent Units

Another type of commonly used RNN layer is the *gated recurrent unit* (GRU).[2] The key differences from the LSTM unit are as follows:

1. The *forget* and *input* gates are replaced by *reset* and *update* gates.
2. There is no *cell state* or *output* gate, only a *hidden state* that is output from the cell.

The hidden state is updated in four steps, as illustrated in Figure 5-12.

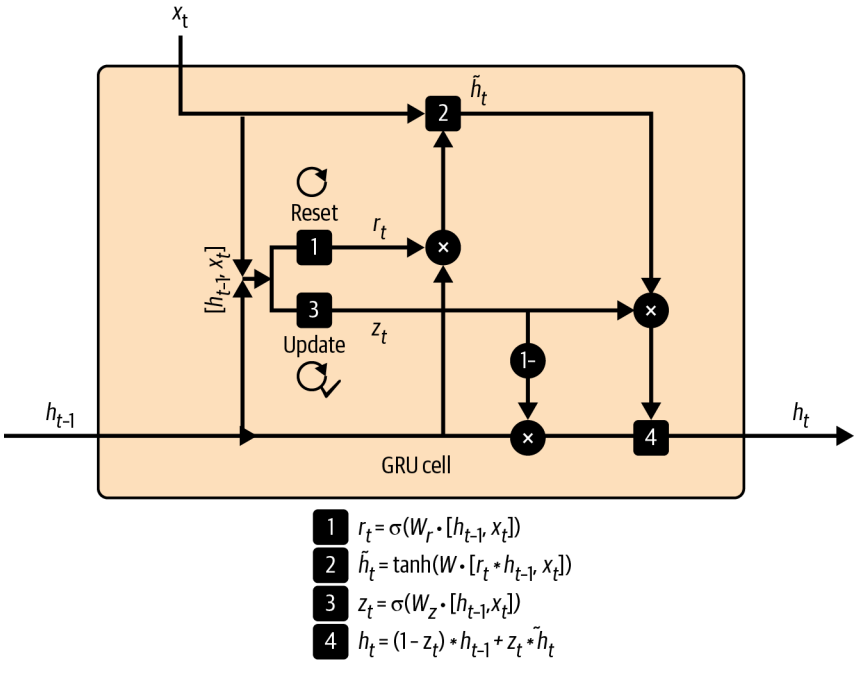

Figure 5-12. A single GRU cell

The process is as follows:

1. The hidden state of the previous timestep, h_{t-1}, and the current word embedding, x_t, are concatenated and used to create the *reset* gate. This gate is a dense layer, with weights matrix W_r and a sigmoid activation function. The resulting vector, r_t, has length equal to the number of units in the cell and stores values between 0 and 1 that determine how much of the previous hidden state, h_{t-1}, should be carried forward into the calculation for the new beliefs of the cell.
2. The reset gate is applied to the hidden state, h_{t-1}, and concatenated with the current word embedding, x_t. This vector is then fed to a dense layer with weights matrix W and a tanh activation function to generate a vector, $\tilde{h}_t$, that stores the new beliefs of the cell. It has length equal to the number of units in the cell and stores values between –1 and 1.
3. The concatenation of the hidden state of the previous timestep, h_{t-1}, and the current word embedding, x_t, are also used to create the *update* gate. This gate is a dense layer with weights matrix W_z and a sigmoid activation. The resulting vector, z_t, has length equal to the number of units in the cell and stores values

between 0 and 1, which are used to determine how much of the new beliefs, $\tilde{h}_t$, to blend into the current hidden state, h_{t-1}.

4. The new beliefs of the cell, $\tilde{h}_t$, and the current hidden state, h_{t-1}, are blended in a proportion determined by the update gate, z_t, to produce the updated hidden state, h_t, that is output from the cell.

Bidirectional Cells

For prediction problems where the entire text is available to the model at inference time, there is no reason to process the sequence only in the forward direction—it could just as well be processed backward. A `Bidirectional` layer takes advantage of this by storing two sets of hidden states: one that is produced as a result of the sequence being processed in the usual forward direction and another that is produced when the sequence is processed backward. This way, the layer can learn from information both preceding and succeeding the given timestep.

In Keras, this is implemented as a wrapper around a recurrent layer, as shown in Example 5-11.

Example 5-11. Building a bidirectional GRU layer

```
layer = layers.Bidirectional(layers.GRU(100))
```

Hidden State

The hidden states in the resulting layer are vectors of length equal to double the number of units in the wrapped cell (a concatenation of the forward and backward hidden states). Thus, in this example the hidden states of the layer are vectors of length 200.

So far, we have only applied autoregressive models (LSTMs) to text data. In the next section, we will see how autoregressive models can also be used to generate images.

PixelCNN

In 2016, van den Oord et al.[3] introduced a model that generates images pixel by pixel by predicting the likelihood of the next pixel based on the pixels before it. The model is called *PixelCNN*, and it can be trained to generate images autoregressively.

There are two new concepts that we need to introduce to understand the PixelCNN—*masked convolutional layers* and *residual blocks*.

Running the Code for This Example

The code for this example can be found in the Jupyter notebook located at *notebooks/05_autoregressive/02_pixelcnn/pixelcnn.ipynb* in the book repository.

The code has been adapted from the excellent PixelCNN tutorial created by ADMoreau, available on the Keras website.

Masked Convolutional Layers

As we saw in Chapter 2, a convolutional layer can be used to extract features from an image by applying a series of filters. The output of the layer at a particular pixel is a weighted sum of the filter weights multiplied by the preceding layer values over a small square centered on the pixel. This method can detect edges and textures and, at deeper layers, shapes and higher-level features.

Whilst convolutional layers are extremely useful for feature detection, they cannot directly be used in an autoregressive sense, because there is no ordering placed on the pixels. They rely on the fact that all pixels are treated equally—no pixel is treated as the *start* or *end* of the image. This is in contrast to the text data that we have already seen in this chapter, where there is a clear ordering to the tokens so recurrent models such as LSTMs can be readily applied.

For us to be able to apply convolutional layers to image generation in an autoregressive sense, we must first place an ordering on the pixels and ensure that the filters are only able to see pixels that precede the pixel in question. We can then generate images one pixel at a time, by applying convolutional filters to the current image to predict the value of the next pixel from all preceding pixels.

We first need to choose an ordering for the pixels—a sensible suggestion is to order the pixels from top left to bottom right, moving first along the rows and then down the columns.

We then mask the convolutional filters so that the output of the layer at each pixel is only influenced by pixel values that precede the pixel in question. This is achieved by multiplying a mask of ones and zeros with the filter weights matrix, so that the values of any pixels that are after the target pixel are zeroed.

There are actually two different kinds of masks in a PixelCNN, as shown in Figure 5-13:

- Type A, where the value of the central pixel is masked
- Type B, where the value of the central pixel is *not* masked

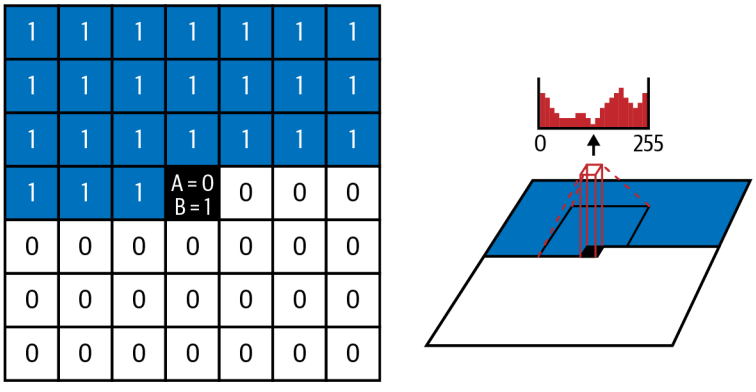

Figure 5-13. Left: a convolutional filter mask; right: a mask applied to a set of pixels to predict the distribution of the central pixel value (source: van den Oord et al., 2016)

The initial masked convolutional layer (i.e., the one that is applied directly to the input image) cannot use the central pixel, because this is precisely the pixel we want the network to guess! However, subsequent layers can use the central pixel because this will have been calculated only as a result of information from preceding pixels in the original input image.

We can see in Example 5-12 how a `MaskedConvLayer` can be built using Keras.

Example 5-12. A `MaskedConvLayer` in Keras

```
class MaskedConvLayer(layers.Layer):
    def __init__(self, mask_type, **kwargs):
        super(MaskedConvLayer, self).__init__()
        self.mask_type = mask_type
        self.conv = layers.Conv2D(**kwargs) ❶

    def build(self, input_shape):
        self.conv.build(input_shape)
        kernel_shape = self.conv.kernel.get_shape()
        self.mask = np.zeros(shape=kernel_shape) ❷
        self.mask[: kernel_shape[0] // 2, ...] = 1.0 ❸
        self.mask[kernel_shape[0] // 2, : kernel_shape[1] // 2, ...] = 1.0 ❹
        if self.mask_type == "B":
            self.mask[kernel_shape[0] // 2, kernel_shape[1] // 2, ...] = 1.0 ❺

    def call(self, inputs):
        self.conv.kernel.assign(self.conv.kernel * self.mask) ❻
        return self.conv(inputs)
```

❶ The `MaskedConvLayer` is based on the normal `Conv2D` layer.

❷ The mask is initialized with all zeros.

❸ The pixels in the preceding rows are unmasked with ones.

❹ The pixels in the preceding columns that are in the same row are unmasked with ones.

❺ If the mask type is B, the central pixel is unmasked with a one.

❻ The mask is multiplied with the filter weights.

Note that this simplified example assumes a grayscale image (i.e., with one channel). If we have color images, we'll have three color channels that we can also place an ordering on so that, for example, the red channel precedes the blue channel, which precedes the green channel.

Residual Blocks

Now that we have seen how to mask the convolutional layer, we can start to build our PixelCNN. The core building block that we will use is the residual block.

A *residual block* is a set of layers where the output is added to the input before being passed on to the rest of the network. In other words, the input has a *fast-track* route to the output, without having to go through the intermediate layers—this is called a *skip connection*. The rationale behind including a skip connection is that if the optimal transformation is just to keep the input the same, this can be achieved by simply zeroing the weights of the intermediate layers. Without the skip connection, the network would have to find an identity mapping through the intermediate layers, which is much harder.

A diagram of the residual block in our PixelCNN is shown in Figure 5-14.

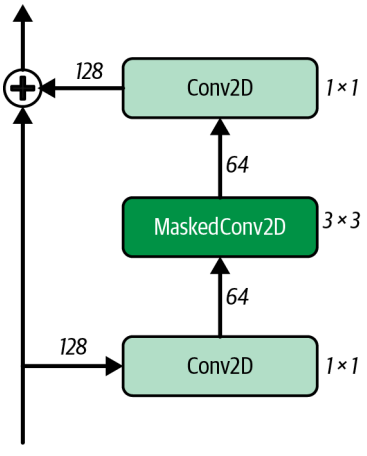

Figure 5-14. A PixelCNN residual block (the numbers of filters are next to the arrows and the filter sizes are next to the layers)

We can build a `ResidualBlock` using the code shown in Example 5-13.

Example 5-13. A `ResidualBlock`

```
class ResidualBlock(layers.Layer):
    def __init__(self, filters, **kwargs):
        super(ResidualBlock, self).__init__(**kwargs)
        self.conv1 = layers.Conv2D(
            filters=filters // 2, kernel_size=1, activation="relu"
        ) ❶
        self.pixel_conv = MaskedConv2D(
            mask_type="B",
            filters=filters // 2,
            kernel_size=3,
            activation="relu",
            padding="same",
        ) ❷
        self.conv2 = layers.Conv2D(
            filters=filters, kernel_size=1, activation="relu"
        ) ❸

    def call(self, inputs):
        x = self.conv1(inputs)
        x = self.pixel_conv(x)
        x = self.conv2(x)
        return layers.add([inputs, x]) ❹
```

❶ The initial `Conv2D` layer halves the number of channels.

❷ The Type B `MaskedConv2D` layer with kernel size of 3 only uses information from five pixels—three pixels in the row above the focus pixel, one to the left, and the focus pixel itself.

❸ The final `Conv2D` layer doubles the number of channels to again match the input shape.

❹ The output from the convolutional layers is added to the input—this is the skip connection.

Training the PixelCNN

In Example 5-14 we put together the whole PixelCNN network, approximately following the structure laid out in the original paper. In the original paper, the output layer is a 256-filter `Conv2D` layer, with softmax activation. In other words, the network tries to re-create its input by predicting the correct pixel values, a bit like an autoencoder. The difference is that the PixelCNN is constrained so that no information from earlier pixels can flow through to influence the prediction for each pixel, due to the way that network is designed, using `MaskedConv2D` layers.

A challenge with this approach is that the network has no way to understand that a pixel value of, say, 200 is very close to a pixel value of 201. It must learn every pixel output value independently, which means training can be very slow, even for the simplest datasets. Therefore, in our implementation, we instead simplify the input so that each pixel can take only one of four values. This way, we can use a 4-filter `Conv2D` output layer instead of 256.

Example 5-14. The PixelCNN architecture

```
inputs = layers.Input(shape=(16, 16, 1)) ❶
x = MaskedConv2D(mask_type="A"
                 , filters=128
                 , kernel_size=7
                 , activation="relu"
                 , padding="same")(inputs)❷

for _ in range(5):
    x = ResidualBlock(filters=128)(x) ❸

for _ in range(2):
    x = MaskedConv2D(
        mask_type="B",
        filters=128,
        kernel_size=1,
        strides=1,
        activation="relu",
```

```
        padding="valid",
    )(x) ❹

out = layers.Conv2D(
    filters=4, kernel_size=1, strides=1, activation="softmax", padding="valid"
)(x) ❺

pixel_cnn = models.Model(inputs, out) ❻

adam = optimizers.Adam(learning_rate=0.0005)
pixel_cnn.compile(optimizer=adam, loss="sparse_categorical_crossentropy")

pixel_cnn.fit(
    input_data
    , output_data
    , batch_size=128
    , epochs=150
) ❼
```

❶ The model `Input` is a grayscale image of size 16 × 16 × 1, with inputs scaled between 0 and 1.

❷ The first Type A `MaskedConv2D` layer with a kernel size of 7 uses information from 24 pixels—21 pixels in the three rows above the focus pixel and 3 to the left (the focus pixel itself is not used).

❸ Five `ResidualBlock` layer groups are stacked sequentially.

❹ Two Type B `MaskedConv2D` layers with a kernel size of 1 act as `Dense` layers across the number of channels for each pixel.

❺ The final `Conv2D` layer reduces the number of channels to four—the number of pixel levels for this example.

❻ The `Model` is built to accept an image and output an image of the same dimensions.

❼ Fit the model—`input_data` is scaled in the range [0, 1] (floats); `output_data` is scaled in the range [0, 3] (integers).

Analysis of the PixelCNN

We can train our PixelCNN on images from the Fashion-MNIST dataset that we encountered in Chapter 3. To generate new images, we need to ask the model to predict the next pixel given all preceding pixels, one pixel at a time. This is a very slow process compared to a model such as a variational autoencoder! For a 32 × 32

grayscale image, we need to make 1,024 predictions sequentially using the model, compared to the single prediction that we need to make for a VAE. This is one of the major downsides to autoregressive models such as a PixelCNN—they are slow to sample from, because of the sequential nature of the sampling process.

For this reason, we use an image size of 16 × 16, rather than 32 × 32, to speed up the generation of new images. The generation callback class is shown in Example 5-15.

Example 5-15. Generating new images using the PixelCNN

```
class ImageGenerator(callbacks.Callback):
    def __init__(self, num_img):
        self.num_img = num_img

    def sample_from(self, probs, temperature):
        probs = probs ** (1 / temperature)
        probs = probs / np.sum(probs)
        return np.random.choice(len(probs), p=probs)

    def generate(self, temperature):
        generated_images = np.zeros(
            shape=(self.num_img,) + (pixel_cnn.input_shape)[1:]
        ) ❶
        batch, rows, cols, channels = generated_images.shape

        for row in range(rows):
            for col in range(cols):
                for channel in range(channels):
                    probs = self.model.predict(generated_images)[
                        :, row, col, :
                    ] ❷
                    generated_images[:, row, col, channel] = [
                        self.sample_from(x, temperature) for x in probs
                    ] ❸
                    generated_images[:, row, col, channel] /= 4 ❹
        return generated_images

    def on_epoch_end(self, epoch, logs=None):
        generated_images = self.generate(temperature = 1.0)
        display(
            generated_images,
            save_to = "./output/generated_img_%03d.png" % (epoch)
        s)

img_generator_callback = ImageGenerator(num_img=10)
```

❶ Start with a batch of empty images (all zeros).

❷ Loop over the rows, columns, and channels of the current image, predicting the distribution of the next pixel value.

❸ Sample a pixel level from the predicted distribution (for our example, a level in the range [0, 3]).

❹ Convert the pixel level to the range [0, 1] and overwrite the pixel value in the current image, ready for the next iteration of the loop.

In Figure 5-15, we can see several images from the original training set, alongside images that have been generated by the PixelCNN.

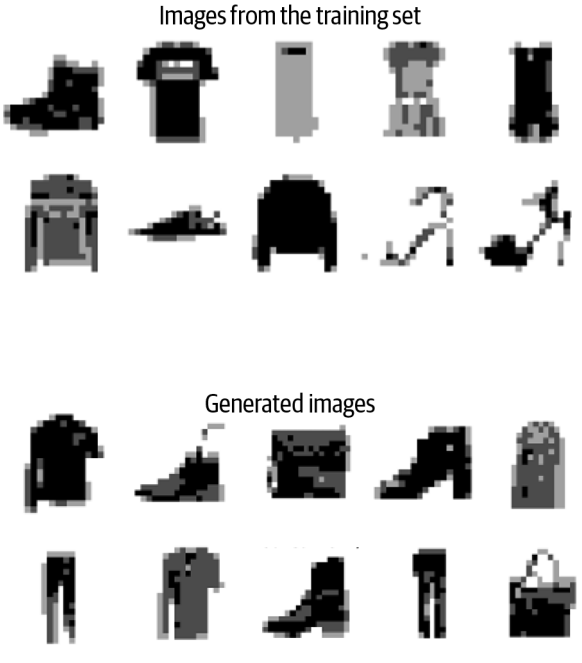

Figure 5-15. Example images from the training set and generated images created by the PixelCNN model

The model does a great job of re-creating the overall shape and style of the original images! It is quite amazing that we can treat images as a series of tokens (pixel values) and apply autoregressive models such as a PixelCNN to produce realistic samples.

As mentioned previously, one of the downsides to autoregressive models is that they are slow to sample from, which is why a simple example of their application is presented in this book. However, as we shall see in Chapter 10, more complex forms of autoregressive model can be applied to images to produce state-of-the-art outputs. In

such cases, the slow generation speed is a necessary price to pay in return for exceptional-quality outputs.

Since the original paper was published, several improvements have been made to the architecture and training process of the PixelCNN. The following section introduces one of those changes—using mixture distributions—and demonstrates how to train a PixelCNN model with this improvement using a built-in TensorFlow function.

Mixture Distributions

For our previous example, we reduced the output of the PixelCNN to just 4 pixel levels to ensure the network didn't have to learn a distribution over 256 independent pixel values, which would slow the training process. However, this is far from ideal—for color images, we wouldn't want our canvas to be restricted to only a handful of possible colors.

To get around this problem, we can make the output of the network a *mixture distribution*, instead of a softmax over 256 discrete pixel values, following the ideas presented by Salimans et al.[4] A mixture distribution is quite simply a mixture of two or more other probability distributions. For example, we could have a mixture distribution of five logistic distributions, each with different parameters. The mixture distribution also requires a discrete categorical distribution that denotes the probability of choosing each of the distributions included in the mix. An example is shown in Figure 5-16.

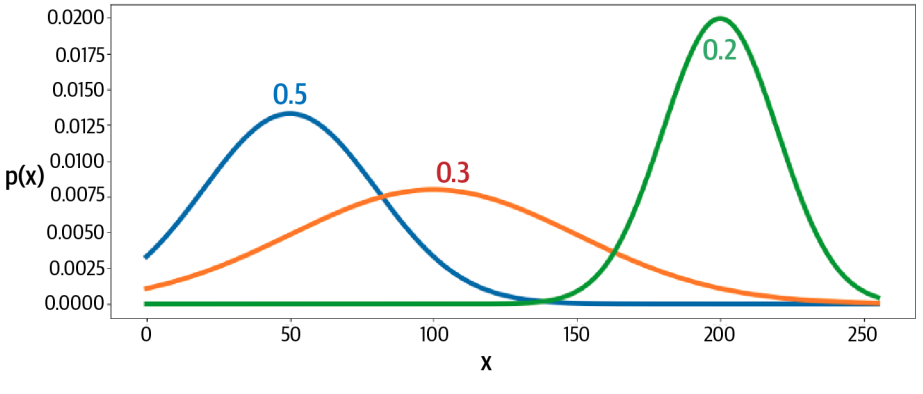

Figure 5-16. A mixture distribution of three normal distributions with different parameters—the categorical distribution over the three normal distributions is `[0.5, 0.3, 0.2]`

To sample from a mixture distribution, we first sample from the categorical distribution to choose a particular subdistribution and then sample from this in the usual way. This way, we can create complex distributions with relatively few parameters.

For example, the mixture distribution in Figure 5-16 only requires eight parameters—two for the categorical distribution and a mean and variance for each of the three normal distributions. This is compared to the 255 parameters that would define a categorical distribution over the entire pixel range.

Conveniently, the TensorFlow Probability library provides a function that allows us to create a PixelCNN with mixture distribution output in a single line. Example 5-16 illustrates how to build a PixelCNN using this function.

Running the Code for This Example

The code for this example can be found in the Jupyter notebook in *notebooks/05_autoregressive/03_pixelcnn_md/pixelcnn_md.ipynb* in the book repository.

Example 5-16. Building a PixelCNN using the TensorFlow function

```
import tensorflow_probability as tfp

dist = tfp.distributions.PixelCNN(
    image_shape=(32, 32, 1),
    num_resnet=1,
    num_hierarchies=2,
    num_filters=32,
    num_logistic_mix=5,
    dropout_p=.3,
) ❶

image_input = layers.Input(shape=(32, 32, 1)) ❷

log_prob = dist.log_prob(image_input)

model = models.Model(inputs=image_input, outputs=log_prob) ❸
model.add_loss(-tf.reduce_mean(log_prob)) ❹
```

❶ Define the PixelCNN as a distribution—i.e., the output layer is a mixture distribution made up of five logistic distributions.

❷ The input is a grayscale image of size $32 \times 32 \times 1$.

❸ The `Model` takes a grayscale image as input and outputs the log-likelihood of the image under the mixture distribution calculated by the PixelCNN.

❹ The loss function is the mean negative log-likelihood over the batch of input images.

The model is trained in the same way as before, but this time accepting integer pixel values as input, in the range [0, 255]. Outputs can be generated from the distribution using the `sample` function, as shown in Example 5-17.

Example 5-17. Sampling from the PixelCNN mixture distribution

```
dist.sample(10).numpy()
```

Example generated images are shown in Figure 5-17. The difference from our previous examples is that now the full range of pixel values is being utilized.

Figure 5-17. Outputs from the PixelCNN using a mixture distribution output

Summary

In this chapter we have seen how autoregressive models such as recurrent neural networks can be applied to generate text sequences that mimic a particular style of writing, and also how a PixelCNN can generate images in a sequential fashion, one pixel at a time.

We explored two different types of recurrent layers—long short-term memory (LSTM) and gated recurrent unit (GRU)—and saw how these cells can be stacked or made bidirectional to form more complex network architectures. We built an LSTM to generate realistic recipes using Keras and saw how to manipulate the temperature of the sampling process to increase or decrease the randomness of the output.

We also saw how images can be generated in an autoregressive manner, using a PixelCNN. We built a PixelCNN from scratch using Keras, coding the masked convolutional layers and residual blocks to allow information to flow through the network so that only preceding pixels could be used to generate the current pixel. Finally, we discussed how the TensorFlow Probability library provides a standalone `PixelCNN` function that implements a mixture distribution as the output layer, allowing us to further improve the learning process.

In the next chapter we will explore another generative modeling family that explicitly models the data-generating distribution—normalizing flow models.

References

1. Sepp Hochreiter and Jürgen Schmidhuber, "Long Short-Term Memory," *Neural Computation* 9 (1997): 1735–1780, *https://www.bioinf.jku.at/publications/older/2604.pdf*.

2. Kyunghyun Cho et al., "Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation," June 3, 2014, *https://arxiv.org/abs/1406.1078*.

3. Aaron van den Oord et al., "Pixel Recurrent Neural Networks," August 19, 2016, *https://arxiv.org/abs/1601.06759*.

4. Tim Salimans et al., "PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications," January 19, 2017, *http://arxiv.org/abs/1701.05517*.