Our `SimpleRnn` has a couple of undesirable features. One is that its entire hidden state is used to update the input every time you call it. The other is that the entire hidden state is overwritten every time you call it. Both of these make it difficult to train; in particular, they make it difficult for the model to learn long-range dependencies.

For this reason, almost no one uses this kind of simple RNN. Instead, they use more complicated variants like the LSTM ("long short-term memory") or the GRU ("gated recurrent unit"), which have many more parameters and use parameterized "gates" that allow only some of the state to be updated (and only some of the state to be used) at each timestep.

There is nothing particularly *difficult* about these variants; however, they involve a great deal more code, which would not be (in my opinion) correspondingly more edifying to read. The code for this chapter on GitHub includes an LSTM implementation. I encourage you to check it out, but it's somewhat tedious and so we won't discuss it further here.

One other quirk of our implementation is that it takes only one "step" at a time and requires us to manually reset the hidden state. A more practical RNN implementation might accept sequences of inputs, set its hidden state to 0s at the beginning of each sequence, and produce sequences of outputs. Ours could certainly be modified to behave this way; again, this would require more code and complexity for little gain in understanding.

Example: Using a Character-Level RNN

The newly hired VP of Branding did not come up with the name *DataSciencester* himself, and (accordingly) he suspects that a better name might lead to more success for the company. He asks you to use data science to suggest candidates for replacement.

One "cute" application of RNNs involves using *characters* (rather than words) as their inputs, training them to learn the subtle language patterns in some dataset, and then using them to generate fictional instances from that dataset.

For example, you could train an RNN on the names of alternative bands, use the trained model to generate new names for fake alternative bands, and then hand-select the funniest ones and share them on Twitter. Hilarity!

Having seen this trick enough times to no longer consider it clever, you decide to give it a shot.

After some digging, you find that the startup accelerator Y Combinator has published a list of its top 100 (actually 101) most successful startups, which seems like a good starting point. Checking the page, you find that the company names all live inside `<b class="h4">` tags, which means it's easy to use your web scraping skills to retrieve them:

```python
from bs4 import BeautifulSoup
import requests

url = "https://www.ycombinator.com/topcompanies/"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')

# We get the companies twice, so use a set comprehension to deduplicate.
companies = list({b.text
                  for b in soup("b")
                  if "h4" in b.get("class", ())})
assert len(companies) == 101
```

As always, the page may change (or vanish), in which case this code won't work. If so, you can use your newly learned data science skills to fix it or just get the list from the book's GitHub site.

So what is our plan? We'll train a model to predict the next character of a name, given the current character *and* a hidden state representing all the characters we've seen so far.

As usual, we'll actually predict a probability distribution over characters and train our model to minimize the `SoftmaxCrossEntropy` loss.

Once our model is trained, we can use it to generate some probabilities, randomly sample a character according to those probabilities, and then feed that character as its next input. This will allow us to *generate* company names using the learned weights.

To start with, we should build a `Vocabulary` from the characters in the names:

```
vocab = Vocabulary([c for company in companies for c in company])
```

In addition, we'll use special tokens to signify the start and end of a company name. This allows the model to learn which characters should *begin* a company name and also to learn when a company name is *finished*.

We'll just use the regex characters for start and end, which (luckily) don't appear in our list of companies:

```
START = "^"
STOP = "$"

# We need to add them to the vocabulary too.
vocab.add(START)
vocab.add(STOP)
```

For our model, we'll one-hot-encode each character, pass it through two `SimpleRnn`s, and then use a `Linear` layer to generate the scores for each possible next character:

```
HIDDEN_DIM = 32  # You should experiment with different sizes!

rnn1 =  SimpleRnn(input_dim=vocab.size, hidden_dim=HIDDEN_DIM)
rnn2 =  SimpleRnn(input_dim=HIDDEN_DIM, hidden_dim=HIDDEN_DIM)
linear = Linear(input_dim=HIDDEN_DIM, output_dim=vocab.size)

model = Sequential([
    rnn1,
    rnn2,
    linear
])
```

Imagine for the moment that we've trained this model. Let's write the function that uses it to generate new company names, using the `sample_from` function from "Topic Modeling":

```
from scratch.deep_learning import softmax

def generate(seed: str = START, max_len: int = 50) -> str:
    rnn1.reset_hidden_state()  # Reset both hidden states
    rnn2.reset_hidden_state()
    output = [seed]            # Start the output with the specified seed

    # Keep going until we produce the STOP character or reach the max length
    while output[-1] != STOP and len(output) < max_len:
        # Use the last character as the input
        input = vocab.one_hot_encode(output[-1])

        # Generate scores using the model
        predicted = model.forward(input)

        # Convert them to probabilities and draw a random char_id
        probabilities = softmax(predicted)
        next_char_id = sample_from(probabilities)

        # Add the corresponding char to our output
        output.append(vocab.get_word(next_char_id))

    # Get rid of START and END characters and return the word
    return ''.join(output[1:-1])
```

At long last, we’re ready to train our character-level RNN. It will take a while!

```
loss = SoftmaxCrossEntropy()
optimizer = Momentum(learning_rate=0.01, momentum=0.9)

for epoch in range(300):
    random.shuffle(companies)  # Train in a different order each epoch.
    epoch_loss = 0             # Track the loss.
    for company in tqdm.tqdm(companies):
        rnn1.reset_hidden_state()  # Reset both hidden states.
        rnn2.reset_hidden_state()
        company = START + company + STOP   # Add START and STOP characters.

        # The rest is just our usual training loop, except that the inputs
        # and target are the one-hot-encoded previous and next characters.
        for prev, next in zip(company, company[1:]):
            input = vocab.one_hot_encode(prev)
            target = vocab.one_hot_encode(next)
            predicted = model.forward(input)
            epoch_loss += loss.loss(predicted, target)
            gradient = loss.gradient(predicted, target)
            model.backward(gradient)
            optimizer.step(model)

    # Each epoch, print the loss and also generate a name.
    print(epoch, epoch_loss, generate())

    # Turn down the learning rate for the last 100 epochs.
    # There's no principled reason for this, but it seems to work.
    if epoch == 200:
        optimizer.lr *= 0.1
```

After training, the model generates some actual names from the list (which isn't surprising, since the model has a fair amount of capacity and not a lot of training data), as well as names that are only slightly different from training names (Scripe, Loinbare, Pozium), names that seem genuinely creative (Benuus, Cletpo, Equite, Vivest), and names that are garbage-y but still sort of word-like (SFitreasy, Sint ocanelp, GliyOx, Doorboronelhav).

Unfortunately, like most character-level-RNN outputs, these are only mildly clever, and the VP of Branding ends up unable to use them.

If I up the hidden dimension to 64, I get a lot more names verbatim from the list; if I drop it to 8, I get mostly garbage. The vocabulary and final weights for all these model sizes are available on the book's GitHub site, and you can use `load_weights` and `load_vocab` to use them yourself.

As mentioned previously, the GitHub code for this chapter also contains an implementation for an LSTM, which you should feel free to swap in as a replacement for the `SimpleRnn`s in our company name model.

For Further Exploration

- NLTK is a popular library of NLP tools for Python. It has its own entire book, which is available to read online.
- gensim is a Python library for topic modeling, which is a better bet than our from-scratch model.

- spaCy is a library for “Industrial Strength Natural Language Processing in Python” and is also quite popular.
- Andrej Karpathy has a famous blog post, “The Unreasonable Effectiveness of Recurrent Neural Networks”, that’s very much worth reading.
- My day job involves building AllenNLP, a Python library for doing NLP research. (At least, as of the time this book went to press, it did.) The library is quite beyond the scope of this book, but you might still find it interesting, and it has a cool interactive demo of many state-of-the-art NLP models.