I found the CBOW model harder to train than the skip-gram one, but I encourage you to try it out.

Recurrent Neural Networks

The word vectors we developed in the previous section are often used as the inputs to neural networks. One challenge to doing this is that sentences have varying lengths: you could think of a 3-word sentence as a `[3, embedding_dim]` tensor and a 10-word sentence as a `[10, embedding_dim]` tensor. In order to, say, pass them to a `Linear` layer, we need to do something about that first variable-length dimension.

One option is to use a `Sum` layer (or a variant that takes the average); however, the *order* of the words in a sentence is usually important to its meaning. To take a common example, "dog bites man" and "man bites dog" are two very different stories!

Another way of handling this is using *recurrent neural networks* (RNNs), which have a *hidden state* they maintain between inputs. In the simplest case, each input is combined with the current hidden state to produce an output, which is then used as the new hidden state. This allows such networks to "remember" (in a sense) the inputs they've seen, and to build up to a final output that depends on all the inputs and their order.

We'll create pretty much the simplest possible RNN layer, which will accept a single input (corresponding to, e.g., a single word in a sentence, or a single character in a word), and which will maintain its hidden state between calls.

Recall that our `Linear` layer had some weights, `w`, and a bias, `b`. It took a vector `input` and produced a different vector as `output` using the logic:

```
output[o] = dot(w[o], input) + b[o]
```

Here we'll want to incorporate our hidden state, so we'll have *two* sets of weights—one to apply to the `input` and one to apply to the previous

`hidden` state:

```
output[o] = dot(w[o], input) + dot(u[o], hidden) + b[o]
```

Next, we'll use the `output` vector as the new value of `hidden`. This isn't a huge change, but it will allow our networks to do wonderful things.

```
from scratch.deep_learning import tensor_apply, tanh

class SimpleRnn(Layer):
    """Just about the simplest possible recurrent layer."""
    def __init__(self, input_dim: int, hidden_dim: int) -> None:
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        self.w = random_tensor(hidden_dim, input_dim, init='xavier')
        self.u = random_tensor(hidden_dim, hidden_dim, init='xavier')
        self.b = random_tensor(hidden_dim)

        self.reset_hidden_state()

    def reset_hidden_state(self) -> None:
        self.hidden = [0 for _ in range(self.hidden_dim)]
```

You can see that we start out the hidden state as a vector of 0s, and we provide a function that people using the network can call to reset the hidden state.

Given this setup, the `forward` function is reasonably straightforward (at least, it is if you remember and understand how our `Linear` layer worked):

```
    def forward(self, input: Tensor) -> Tensor:
        self.input = input              # Save both input and previous
        self.prev_hidden = self.hidden  # hidden state to use in backprop.

        a = [(dot(self.w[h], input) +           # weights @ input
              dot(self.u[h], self.hidden) +     # weights @ hidden
              self.b[h])                        # bias
             for h in range(self.hidden_dim)]

        self.hidden = tensor_apply(tanh, a)  # Apply tanh activation
        return self.hidden                   # and return the result.
```

The `backward` pass is similar to the one in our `Linear` layer, except that it needs to compute an additional set of gradients for the `u` weights:

```python
def backward(self, gradient: Tensor):
    # Backpropagate through the tanh
    a_grad = [gradient[h] * (1 - self.hidden[h] ** 2)
              for h in range(self.hidden_dim)]

    # b has the same gradient as a
    self.b_grad = a_grad

    # Each w[h][i] is multiplied by input[i] and added to a[h],
    # so each w_grad[h][i] = a_grad[h] * input[i]
    self.w_grad = [[a_grad[h] * self.input[i]
                    for i in range(self.input_dim)]
                   for h in range(self.hidden_dim)]

    # Each u[h][h2] is multiplied by hidden[h2] and added to a[h],
    # so each u_grad[h][h2] = a_grad[h] * prev_hidden[h2]
    self.u_grad = [[a_grad[h] * self.prev_hidden[h2]
                    for h2 in range(self.hidden_dim)]
                   for h in range(self.hidden_dim)]

    # Each input[i] is multiplied by every w[h][i] and added to a[h],
    # so each input_grad[i] = sum(a_grad[h] * w[h][i] for h in ...)
    return [sum(a_grad[h] * self.w[h][i] for h in range(self.hidden_dim))
            for i in range(self.input_dim)]
```

And finally we need to override the `params` and `grads` methods:

```python
def params(self) -> Iterable[Tensor]:
    return [self.w, self.u, self.b]

def grads(self) -> Iterable[Tensor]:
    return [self.w_grad, self.u_grad, self.b_grad]
```

WARNING

This “simple” RNN is so simple that you probably shouldn’t use it in practice.

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.