PREV
5. Convolutional Neural Networks

NEXT
7. PyTorch

Chapter 6. Recurrent Neural Networks

In this chapter, we'll cover recurrent neural networks (RNNs), a class of neural network architectures meant for handling sequences of data. The neural networks we've seen so far treated each batch of data they received as a set of independent observations; there was no notion of some of the MNIST digits arriving before or after the other digits, in either the fully connected neural networks we saw in Chapter 4 or the convolutional neural networks we saw in Chapter 5. Many kinds of data, however, are intrinsically ordered, whether time series data, which one might deal with in an industrial or financial context, or language data, in which the characters, words, sentences, and so on are ordered. Recurrent neural networks are designed to learn how to take in *sequences* of such data and return a correct prediction as output, whether that correct prediction is of the price of a financial asset on the following day or of the next word in a sentence.

Dealing with ordered data will require three kinds of changes from the fully connected neural networks we saw in the first few chapters. First, it will involve "adding a new dimension" to the `ndarray`s we feed our neural networks. Previously, the data we fed our neural networks was intrinsically two-dimensional—each `ndarray` had one dimension representing the number of observations and another representing the number of features;[1] another way to think of this is that *each observation* was a one-dimensional vector. With recurrent neural networks, each input will still have a dimension representing the number of observations, but each observation will be represented as a two-dimensional `ndarray`: one dimension will represent the length of the sequence of data, and a second dimension will represent the number of features present at each sequence element. The overall input to an RNN will thus be a three-dimensional `ndarray` of shape `[batch_size, sequence_length, num_features]`—a batch of sequences.

Second, of course, to deal with this new three-dimensional input we'll have to use a new kind of neural network architecture, which will be the main focus of this chapter. The third change, however, is where we'll start our discussion in this chapter: we'll have to use a completely different framework with different abstractions to deal with this new form of data. Why? In the cases of both fully connected and convolutional neural networks, each "operation," even if it in fact `ndarray` as input and produced one `ndarray` as an output (possibly using another `ndarray` representing the operation's parameters as part of these computations). As it turns out, recurrent neural networks cannot be implemented in this way. Before reading further to find out why, take some time to think about it: what characteristics of a neural network architecture would cause the framework we've built so far to break down? While the answer is illuminating, the full solution involves concepts that go deep into implementation details and are beyond the scope of this book.[2] To begin to unpack this, let's reveal a key limitation of the framework we've been using so far.

The Key Limitation: Handling Branching

It turns out that our framework couldn't train models with computational graphs like those depicted in Figure 6-1.

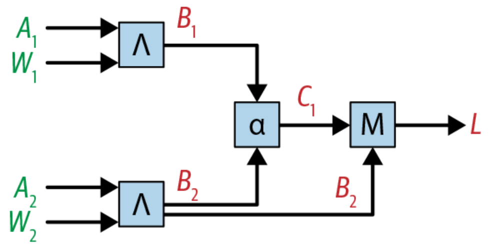

Figure 6-1. The computational graph that causes our Operation framework to fail: the same quantity is repeated multiple times during the forward pass, meaning that we can't simply send gradients backward in sequence during the backward pass as before

What's wrong with this? Translating the forward pass into code seems fine (note that we have written `Add` and `Multiply` operations here for illustration purposes only):

```
a1 = torch.randn(3,3)
w1 = torch.randn(3,3)

a2 = torch.randn(3,3)
w2 = torch.randn(3,3)

w3 = torch.randn(3,3)

# operations
wm1 = WeightMultiply(w1)
wm2 = WeightMultiply(w2)
add2 = Add(2, 1)
mult3 = Multiply(2, 1)

b1 = wm1.forward(a1)
b2 = wm2.forward(a2)
c1 = add2.forward((b1, b2))
L = mult3.forward((c1, b2))
```

The trouble begins when we start the backward pass. Let's say we want to use our usual chain rule logic to calculate the derivative of `L` with respect to `w1`. Previously, we would simply call `backward` on each operation in reverse order. Here, because of the *reuse of* `b2` *during the forward pass*, that approach doesn't work. If we began by calling `backward` on `mult3`, for example, we would have gradients for each of its inputs, `c1` and `b2`. However, if we then called `backward` on `add2`, we couldn't just feed in the gradient for `c1`: we would have to also feed in the gradient for `b2` somehow, since this also affects the loss `L`. So, to properly

```
c1_grad, b2_grad_1 = mult3.backward(L_grad)

b1_grad, b2_grad_2 = add2.backward(c1_grad)

# combine these gradients to reflect the fact that b2 is used twice on
# forward pass
b2_grad = b2_grad_1 + b2_grad_2

a2_grad = wm2.backward(b2_grad)

a1_grad = wm1.backward(b1_grad)
```

At this point we might as well skip using `Operation`s entirely; we can simply save all of the quantities we computed on the forward pass and reuse them on the backward pass, as we were doing in Chapter 2! We could *always* code arbitrarily complicated neural networks by manually defining the individual computations to be done on the forward and backward passes of the network, just as we wrote out the 17 individual operations involved in the backward pass of a two-layer neural network in Chapter 2 (indeed, we'll do something like this later in this chapter inside "RNN cells"). What we were trying to do with `Operation`s was to build a flexible framework that let us describe a neural network in high-level terms and have all of the low-level computations "just work." While this framework illustrated many key concepts about neural networks, we now see its limitations.

There is an elegant solution to this problem: automatic differentiation, a completely different way of implementing neural networks.[3] We'll cover just enough of this concept here to give you an idea of how it works but won't go further than that building a full-featured automatic differentiation framework would take several chapters just on its own. Furthermore, we'll see how to *use* a high-performance automatic differentiation framework in the next chapter when we cover PyTorch. Still, automatic differentiation is an important enough concept to understand from first principles that before we get into RNNs, we'll design a basic framework for it and show how it solves the problem with reusing objects during the forward pass described in the preceding example.

Automatic Differentiation

As we've seen, there are some neural network architectures for which the `Operation` framework we've been using so far can't easily compute the gradients of the output with respect to the inputs, as we have to do to be able to train our models. Automatic differentiation allows us to compute these gradients via a completely different route: rather than the `Operation`s being the atomic units that make up the network, we define a class that wraps around the data itself and allows the data to keep track of the operations performed on it, so that the data can continually accumulate gradients as it is involved in different operations. To understand better how this "gradient accumulation" would work, let's start to code it up.[4]

Coding Up Gradient Accumulation

To automatically keep track of gradients, we have to overwrite the Python methods that perform basic operations on our data. In Python, using operators such as **+** or **–** actually calls underlying hidden methods such as `__add__` and `__sub__`. For example, here's how that works with **+**:

```
a = array([3,3])
print("Addition using '__add__':", a.__add__(4))
print("Addition using '+':", a + 4)
```

```
Addition using '__add__': [7 7]
Addition using '+': [7 7]
```

```
Numberable = Union[float, int]

def ensure_number(num: Numberable) -> NumberWithGrad:
    if isinstance(num, NumberWithGrad):
        return num
    else:
        return NumberWithGrad(num)

class NumberWithGrad(object):

    def __init__(self,
                 num: Numberable,
                 depends_on: List[Numberable] = None,
                 creation_op: str = ''):
        self.num = num
        self.grad = None
        self.depends_on = depends_on or []
        self.creation_op = creation_op

    def __add__(self,
                other: Numberable) -> NumberWithGrad:
        return NumberWithGrad(self.num + ensure_number(other).num,
                              depends_on = [self, ensure_number(othe
                              creation_op = 'add')

    def __mul__(self,
                other: Numberable = None) -> NumberWithGrad:

        return NumberWithGrad(self.num * ensure_number(other).num,
                              depends_on = [self, ensure_number(othe
                              creation_op = 'mul')

    def backward(self, backward_grad: Numberable = None) -> None:
        if backward_grad is None: # first time calling backward
            self.grad = 1
        else:
            # These lines allow gradients to accumulate.
            # If the gradient doesn't exist yet, simply set it equal
            # to backward_grad
            if self.grad is None:
                self.grad = backward_grad
            # Otherwise, simply add backward_grad to the existing grad
            else:
                self.grad += backward_grad

        if self.creation_op == "add":
            # Simply send backward self.grad, since increasing either
            # elements will increase the output by that same amount
            self.depends_on[0].backward(self.grad)
            self.depends_on[1].backward(self.grad)

        if self.creation_op == "mul":

            # Calculate the derivative with respect to the first eleme
            new = self.depends_on[1] * self.grad
            # Send backward the derivative with respect to that elemen
            self.depends_on[0].backward(new.num)

            # Calculate the derivative with respect to the second elem
            new = self.depends_on[0] * self.grad
            # Send backward the derivative with respect to that elemen
            self.depends_on[1].backward(new.num)
```

There's a lot going on here, so let's unpack this `NumberWithGrad` class and see how it works. Recall that the goal of such a class is to be able to write simple operations and have the gradients be computed automatically; for example, suppose we write:

```
a = NumberWithGrad(3)

b = a * 4
c = b + 5
```

At this point, how much will increasing a by ϵ increase the value of c? It should

```
c.backward()
```

then, without writing a `for` loop to iterate through the `Operation`s or anything, we can write:

```
print(a.grad)
```

```
4
```

How does this work? The fundamental insight incorporated into the previous class is that every time the `+` or `*` operations are performed on a `NumberWithGrad`, a new `NumberWithGrad` is created, with the first `NumberWithGrad` as a dependency. Then, when `backward` is called on a `NumberWithGrad`, like it is called on `c` previously, all the gradients for all of the `NumberWithGrad`s used to create `c` are automatically calculated. So indeed, not only was the gradient for `a` calculated, but so was the gradient for `b`:

```
print(b.grad)
```

```
1
```

The real beauty of this framework, however, is that it allows the `NumberWithGrad`s to *accumulate* gradients and thus be reused multiple times during a series of computations, and we still end up with the correct gradient. We'll illustrate this with the same series of operations that stumped us before, using a `NumberWithGrad` multiple times in a series of computations, and then unpack how it works in detail.

AUTOMATIC DIFFERENTIATION ILLUSTRATION

Here's the series of computations in which `a` is reused multiple times:

```
a = NumberWithGrad(3)

b = a * 4
c = b + 3
d = c * (a + 2)
```

We can work out that if we do these operations, $d = 75$, but as we know, the real question is: how much will increasing the value of `a` increase the value of `d`? We can first work out the answer to this question mathematically. We have:

$$d = (4a + 3) \times (a + 2) = 4a^2 + 11a + 6$$

So, using the power rule from calculus:

$$\frac{\partial d}{\partial a} = 8a + 11$$

For $a = 3$, therefore, the value of this derivative should be $8 \times 3 + 11 = 35$. Confirming this numerically:

```
def forward(num: int):
    b = num * 4
    c = b + 3
    return c * (num + 2)

print(round(forward(3.01) - forward(2.99)) / 0.02), 3)
```

```
35.0
```

```
a = NumberWithGrad(3)

b = a * 4
c = b + 3
d = (a + 2)
e = c * d
e.backward()

print(a.grad)
```

```
35
```

EXPLAINING WHAT HAPPENED

As we can see, the goal with automatic differentiation is to make the *data objects themselves*—numbers, `ndarrays`, `Tensors`, and so on—the fundamental units of analysis, rather than the `Operation`s as before.

All automatic differentiation techniques have the following in common:

- Each technique includes a class that wraps around the actual data being computed. Here, we wrap `NumberWithGrad` around `float`s and `int`s; in PyTorch, for example, the analogous class is called `Tensor`.
- Common operations such as adding, multiplying, and matrix multiplication are redefined so that they always return members of this class; in the preceding case, we ensure the addition of *either* a `NumberWithGrad` and a `NumberWithGrad` *or* a `NumberWithGrad` and a `float` or an `int`.
- The `NumberWithGrad` class must contain information on how to compute gradients, given what happens on the forward pass. Previously, we did this by including a `creation_op` argument in the class that simply recorded how the `NumberWithGrad` was created.
- On the backward pass, gradients are passed backward using the underlying data type, not the wrapper. Here, this means gradients are of type `float` and `int`, not `NumberWithGrad`.
- As mentioned at the start of this section, automatic differentiation allows us to reuse quantities computed during the forward pass—in the preceding example, we use `a` twice with no problem. The key for allowing this is these lines:

  ```
  if self.grad is None:
      self.grad = backward_grad
  else:
      self.grad += backward_grad
  ```

  These lines state that upon receiving a new gradient, `backward_grad`, a `NumberWithGrad` should either initialize the gradient of `NumberWithGrad` to be this value, or simply add the value to the `NumberWithGrad`'s existing gradient. This is what allows the `NumberWithGrad`s to accumulate gradients as the relevant objects are reused in the model.

That's all we'll cover of automatic differentiation. Let's now turn to the model structure that motivated this digression, since it requires certain quantities to be reused during the forward pass to make predictions.

Motivation for Recurrent Neural Networks

As we discussed at the beginning of this chapter, recurrent neural networks are designed to handle data that appears in sequences: instead of each observation

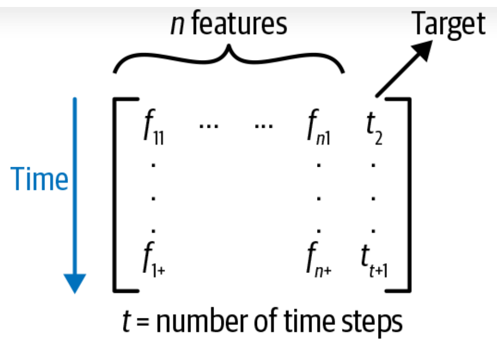

Figure 6-2. Sequential data: at each of the t time steps we have n features

In the following few sections, we'll explain how RNNs accommodate data of this form, but first let's try to understand why we need them. What would be the limitations of simply using a normal feed-forward neural network to deal with this kind of data? One way would be to represent each time step as an independent set of features. For example, one observation could have the features from time `t = 1` with the value of the target from time `t = 2`, the next observation could have the features from time `t = 2` with the value of the target from time `t = 3`, and so on. If we wanted to use data from *multiple* time steps to make each prediction rather than data from just one time step, we could use the features from `t = 1` and `t = 2` to predict the target at `t = 3`, the features from `t = 2` and `t = 3` to predict the target at `t = 4`, and so on.

However, treating each time step as independent ignores the fact that the data is ordered sequentially. How would we ideally want to use the sequential nature of the data to make better predictions? The solution would look something like this:

1. Use features from time step `t = 1` to make predictions for the corresponding target at `t = 1`.
2. Use features from time step `t = 2` *as well as the information from* `t = 1`*, including the value of the target at* `t = 1`, to make predictions for `t = 2`.
3. Use features from time step `t = 3` *as well as the accumulated information from* `t = 1` *and* `t = 2` to make predictions at `t = 3`.
4. And so on, at each step using the information from all prior time steps to make a prediction.

To do this, it seems that we'd want to pass our data through the neural network one sequence element at a time, with the data from the first time step being passed through first, then the data from the next time step, and so on. In addition, we'll want our neural network to "accumulate information" about what it has seen before as the new sequence elements are passed through. We'll spend the rest of this chapter discussing in detail precisely how recurrent neural networks do this. As we'll see, while there are several variants of recurrent neural networks, they all share a common underlying structure in the way they process data sequentially; we'll spend most of our time discussing this structure and at the end discuss the ways in which the variants differ.

Introduction to Recurrent Neural Networks

Let's start our discussion of RNNs by reviewing, at a high level, how data is passed through a "feed-forward" neural network. In this type of network, data is passed forward through a series of *layers*. For a single observation, the output of combinations of these representations, or “features of features” of the original features, and so on for subsequent layers in the network. After each forward pass, then, the network contains in the outputs of each of its layers many representations of the original observation. This is encapsulated in Figure 6-3.

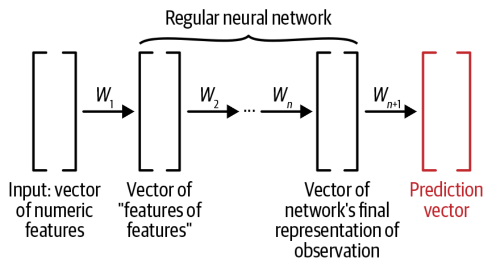
Regular neural network
W_1
W_2
W_n
W_{n+1}
Input: vector of numeric features
Vector of "features of features"
Vector of network's final representation of observation
Prediction vector

Figure 6-3. A regular neural network passing an observation forward and transforming it into different representations after each layer

When the next set of observations is passed through the network, however, these representations are discarded; the key innovation of recurrent neural networks and all of their variants is to *pass these representations back into the network* along with the next set of observations. Here’s how that procedure would look:

1. In the first time step, `t = 1`, we would pass through the observation from the first time step (along with randomly initialized representations, perhaps). We would output a prediction for `t = 1`, along with representations at each layer.

2. In the next time step, we would pass through the observation from the second time step, `t = 2`, along with the representations computed during the first time step (which, again, are just the outputs of the neural network’s layers), and combine these somehow (it is in this combining step that the variants of RNNs we’ll learn about differ). We would use these two pieces of information to output a prediction for `t = 2` as well as the *updated* representations at each layer, which are now a function of the inputs passed in at both `t = 1` *and* `t = 2`.

3. In the third time step, we would pass through the observation from `t = 3`, as well as the representations that now incorporate the information from `t = 1` and `t = 2`, and use this information to make predictions for `t = 3`, as well as additional updated representations at each layer, which now incorporate information from time steps 1–3.

This process is depicted in Figure 6-4.

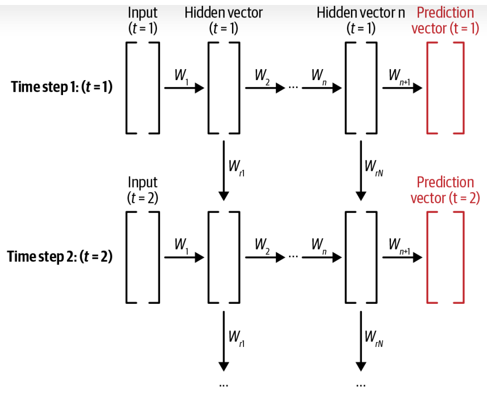

Figure 6-4. Recurrent neural networks pass the representations of each layer forward into the next time step

We see that each layer has a representation that is "persistent," getting updated over time as new observations are passed through. Indeed, this fact is why RNNs are not amenable to the `Operation` framework we've written for the prior chapters: the `ndarray` that represents this state for each layer is continually updated and reused many times in order to make one set of predictions for a single sequence of data using an RNN. Because we can't use the framework from the prior chapter, we'll have to reason from first principles about what classes to build to handle RNNs.

The First Class for RNNs: RNNLayer

Based on the description of how we want RNNs to work, we know at the very least that we'll need an `RNNLayer` class that passes a sequence of data forward one sequence element at a time. Let's now get into the details of how such a class should work. As we've mentioned in this chapter, RNNs will deal with data in which each observation is two-dimensional, with dimensions `(sequence_length, num_features)`; and since it is always more efficient computationally to pass data forward in batches, `RNNLayer` will have to take in three-dimensional `ndarray`s, of size `(batch_size, sequence_length, num_features)`. I explained in the prior section, however, that we want to feed our data through our `RNNLayer`s one sequence element at a time; how can we do this if our input, `data`, is `(batch_size, sequence_length, num_features)`? Here's how:

1. Select a two-dimensional array from the second axis, starting with `data[:, 0, :]`. This `ndarray` will have shape `(batch_size, num_features)`.
2. Initialize a "hidden state" for the `RNNLayer` that will continually get updated with each sequence element passed in, this time of shape `(batch_size, hidden_size)`. This `ndarray` will represent the layer's "accumulated information" about the data that has been passed in during the prior time steps.
3. Pass these two `ndarray`s forward through the first time step in this layer. We'll end up designing `RNNLayer` to output `ndarray`s of different dimensionality than the inputs, just like regular `Dense` layers can, so the output will be of shape `(batch_size, num_outputs)`. In addition, update the neural network's representation for each observation: at each

4. Select the next two-dimensional array from `data`: `data[:, 1, :]`.

5. Pass this data, as well as the values of the RNN's representations outputted at the first time step, into the second time step at this layer to get another output of shape `(batch_size, num_outputs)`, as well as updated representations of shape `(batch_size, hidden_size)`.

6. Continue until all `sequence_length` time steps have been passed through the layer. Then concatenate all the results together to get an output from that layer of shape `(batch_size, sequence_length, num_outputs)`.

This gives us an idea of how our `RNNLayer`s should work—and we'll solidify this understanding when we code it up—but it also hints that we'll need another class to handle receiving the data and updating the layer's hidden state at each time step. For this we'll use the `RNNNode`, the next class we'll cover.

The Second Class for RNNs: RNNNode

Based on the description from the prior section, an `RNNNode` should have a `forward` method with the following inputs and outputs:

- Two `ndarray`s as inputs:
  - One for the data inputs to the network, of shape `[batch_size, num_features]`
  - One for the representations of the observations at that time step, of shape `[batch_size, hidden_size]`
- Two `ndarray`s as outputs:
  - One for the outputs of the network at that time step, or shape `[batch_size, num_outputs]`
  - One for the *updated* representations of the observations at that time step, of shape: `[batch_size, hidden_size]`

Next, we'll show how the two classes, `RNNNode` and `RNNLayer`, fit together.

Putting These Two Classes Together

The `RNNLayer` class will wrap around a list of `RNNNode`s and will (at least) contain a `forward` method that has the following inputs and outputs:

- Input: a batch of sequences of observations of shape `[batch_size, sequence_length, num_features]`
- Output: the neural network output of those sequences of shape `[batch_size, sequence_length, num_outputs]`

Figure 6-5 shows the order that data would move forward through an RNN with two `RNNLayer`s with five `RNNNode`s each. At each time step, inputs initially of dimension `feature_size` are passed successively forward through the first `RNNNode` in each `RNNLayer`, with the network ultimately outputting a prediction at that time step of dimension `output_size`. In addition, each `RNNNode` passes a "hidden state" forward to the next `RNNNode` within each layer. Once data from each of the five time steps has been passed forward through all the layers, we will have a final set of predictions of shape `(5, output_size)`, where `output_size` should be the same dimension as the targets. These predictions

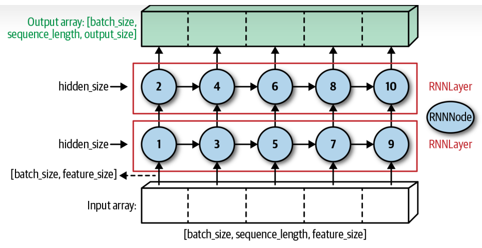
Output array: [batch_size, sequence_length, output_size]
hidden_size
hidden_size
RNNLayer
RNNNode
RNNLayer
[batch_size, feature_size]
Input array:
[batch_size, sequence_length, feature_size]

Figure 6-5. The order in which data would flow through an RNN with two layers that was designed to process sequences of length 5

Alternatively, data could flow through the RNN in the order shown in Figure 6-6. Whatever the order, the following must occur:

- Each layer needs to process its data at a given time step before the next layer—for example, in Figure 6-5, 2 can't happen before 1, and 4 can't happen before 3.
- Similarly, each layer has to process all of its time steps in order—in Figure 6-5, for example, 4 can't happen before 2, and 3 can't happen before 1.
- The last layer has to output dimension `feature_size` for each observation.

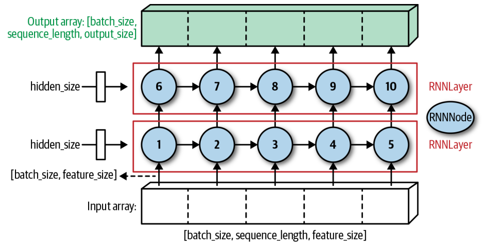
Output array: [batch_size, sequence_length, output_size]
hidden_size
hidden_size
RNNLayer
RNNNode
RNNLayer
[batch_size, feature_size]
Input array:
[batch_size, sequence_length, feature_size]

Figure 6-6. Another order in which data could flow through the same RNN during its forward pass

This all covers how the forward pass through an RNN would work. What about the backward pass?

The Backward Pass

Backpropagation through recurrent neural networks is often described as a separate algorithm called "backpropagation through time." While this does indeed describe what happens during backpropagation, it makes things sound a lot more complicated than they are. Keeping in mind the explanation of how data flows forward through an RNN, we can describe what happens on the backward pass this way: we pass data backward through the RNN by passing gradients backward through the network in reverse of the order that we passed inputs forward on the forward pass—which, indeed, is the same thing we do in regular feed-forward networks.

Looking at the diagrams in Figures 6-5 and 6-6, on the forward pass:

2. These inputs are broken up into the individual `sequence_length` elements and passed into the network one at a time.

3. Each element gets passed through all the layers, ultimately getting transformed into an output of size `output_size`.

4. At the same time, the layer passes the hidden state forward into the layer's computation at the next time step.

5. This continues for all `sequence_length` time steps, resulting in a total output of size `(output_size, sequence_length)`.

Backpropagation simply works the same way, but in reverse:

1. We start with a *gradient* of shape `[output_size, sequence_length]`, representing how much each element of the output (also of size `[output_size, sequence_length]`) ultimately impacts the loss computed for that batch of observations.

2. These gradients are broken up into the individual `sequence_length` elements and passed *backward* through the layers *in reverse order*.

3. The gradient for an individual element is passed backward through all the layers.

4. At the same, the layers pass the *gradient of the loss with respect to the hidden state at that time step* backward into the layers' computations at the prior time steps.

5. This continues for all `sequence_length` time steps, until the gradients have been passed backward to every layer in the network, thus allowing us to compute the gradient of the loss with respect to each of the weights, just as we do in the case of regular feed-forward networks.

This parallelism between the backward and forward pass is highlighted in Figure 6-7, which shows how data flows through an RNN during the backward pass. You'll notice, of course, that it is the same as Figure 6-5 but with the arrows reversed and the numbers changed.

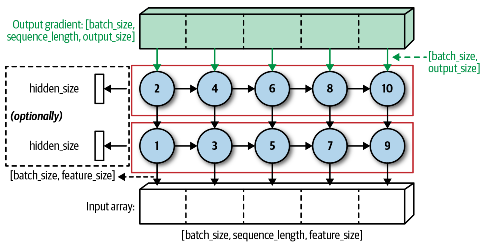
Output gradient: [batch_size, sequence_length, output_size]
[batch_size, output_size]
hidden_size
(optionally)
hidden_size
2
4
6
8
10
1
3
5
7
9
[batch_size, feature_size]
Input array:
[batch_size, sequence_length, feature_size]

Figure 6-7. In the backward pass, RNNs pass data in the opposite direction of the way the data is passed during the forward pass

This highlights that, at a high level, the forward and backward passes for an `RNNLayer` are very similar to those of a layer in a normal neural network: they both receive `ndarray`s of a certain shape as input, output `ndarray`s of another shape, and on the backward pass receive an output gradient of the same shape as their output and produce an input gradient of the same shape as their input. There is a key difference in the way the weight gradients are handled in `RNNLayer`s versus other layers, however, so we'll briefly cover that before we shift to coding this all up.

In recurrent neural networks, just as in regular neural networks, each layer will have *one set of weights*. That means that the same set of weights will affect the layer's output at all `sequence_length` time steps; during backpropagation, therefore, the same set of weights will receive `sequence_length` different gradients. For example, in the circle labeled "1" in the backpropagation shown in Figure 6-7, the second layer will receive a gradient for the last time step, while in the circle labeled "3," the layer will receive a gradient for the second-to-last time step; both of these will be driven by the same set of weights. Thus, during backpropagation, we'll have to *accumulate gradients* for the weights over a series of time steps, which means that however we choose to store the weights, we'll have to update their gradients using something like the following:

```
weight_grad += grad_from_time_step
```

This is different from the `Dense` and `Conv2D` layers, in which we just stored the parameters in a `param_grad` argument.

We've laid out how RNNs work and the classes we want to build to implement them; now let's start figuring out the details.

RNNs: The Code

Let's start with several of the ways the implementation of RNNs will be similar to that of the other neural networks we've covered in this book:

1. An RNN still passes data forward through a series of layers, which send outputs forward on the forward pass and gradients backward on the backward pass. Thus, for example, whatever the equivalent of our `NeuralNetwork` class ends up being will still have a list of `RNNLayer`s as a `layers` attribute, and the forward pass will consist of code like:

```
def forward(self, x_batch: ndarray) -> ndarray:

    assert_dim(ndarray, 3)

    x_out = x_batch
    for layer in self.layers:
        x_out = layer.forward(x_out)

    return x_out
```

2. The `Loss` for RNNs is the same as before: an `ndarray output` is produced by the last `Layer` and compared with `y_batch`, a single value is computed, and a gradient of this value with respect to the input to the `Loss` is returned with the same shape as `output`. We'll have to modify the softmax function to work appropriately with `ndarray`s of shape `[batch_size, sequence_length, feature_size]`, but we can handle that.

3. The `Trainer` is mostly the same: we cycle through our training data, selecting batches of input data and batches of output data, and continually feed them through our model, producing loss values that tell us whether our model is learning and updating the weights after each batch has been fed through. Speaking of which…

4. Our `Optimizer` remains the same as well. As we'll see, we'll have to update how we extract the `params` and `param_grads` at each time step, but the "update rules" (which we captured in the `_update_rule` function in our class) remain the same.

The `Layer`s themselves are where things get interesting.

Previously, we gave `Layer`s a set of `Operation`s that passed data forward and sent gradients backward. `RNNLayer`s will be completely different; they now must maintain a "hidden state" that continually gets updated as new data gets fed in and gets "combined" with the data somehow at each time step. How *exactly* should this work? We can use Figures 6-5 and 6-6 as guides here: they suggest that each `RNNLayer` should have a `List` of `RNNNode`s as an attribute, and then each sequence element from the layer's `input` should get passed through each `RNNNode`, one element at a time. Each `RNNNode` will take in this sequence element, as well as the "hidden state" for that layer, and produce an output for the layer at that time step as well as updating the layer's hidden state.

To clarify all this, let's dive in and start coding it up: we'll cover, in order, how an `RNNLayer` should be initialized, how it should send data forward during the forward pass, and how it should send data backward during the backward pass.

INITIALIZATION

Each `RNNLayer` will start with:

- An `int` `hidden_size`
- An `int` `output_size`
- An `ndarray` `start_H` of shape `(1, hidden_size)`, representing the layer's hidden state

In addition, just like in regular neural networks, we'll set `self.first = True` when we initialize the layer; the first time we pass data into the `forward` method we'll pass the `ndarray` we receive into an `_init_params` method, initialize the parameters, and set `self.first = False`.

With our layer initialized, we are ready to describe how to send data forward.

THE FORWARD METHOD

The bulk of the `forward` method will consist of taking in an `ndarray` `x_seq_in` of shape `(batch_size, sequence_length, feature_size)` and feeding it through the layer's `RNNNode`s in sequence. In the following code, `self.nodes` are the `RNNNode`s for the layer, and `H_in` is the hidden state for the layer:

```
sequence_length = x_seq_in.shape[1]

x_seq_out = np.zeros((batch_size, sequence_length, self.output_size))

for t in range(sequence_length):

    x_in = x_seq_in[:, t, :]

    y_out, H_in = self.nodes[t].forward(x_in, H_in, self.params)

    x_seq_out[:, t, :] = y_out
```

One note on the hidden state `H_in`: the hidden state for an `RNNLayer` is typically represented in a vector, but the operations in each `RNNNode` require the hidden state to be an `ndarray` of size `(batch_size, hidden_size)`. So, at the beginning of each forward pass, we simply "repeat" the hidden state:

```
batch_size = x_seq_in.shape[0]

H_in = np.copy(self.start_H)

H_in = np.repeat(H_in, batch_size, axis=0)
```

```
self.start_H = H_in.mean(axis=0, keepdims=True)
```

Also, we can see from this code that an `RNNNode` will have to have a `forward` method that takes in two arrays of shapes:

- `(batch_size, feature_size)`
- `(batch_size, hidden_size)`

and returns two arrays of shapes:

- `(batch_size, output_size)`
- `(batch_size, hidden_size)`

We'll cover `RNNNode`s (and their variants) in the next section. But first let's cover the `backward` method for the `RNNLayer` class.

THE BACKWARD METHOD

Since the `forward` method outputted `x_seq_out`, the `backward` method will receive a gradient of the same shape as `x_seq_out` called `x_seq_out_grad`. Moving in the opposite direction from the `forward` method, we feed this gradient *backward* through the `RNNNode`s, ultimately returning `x_seq_in_grad` of shape `(batch_size, sequence_length, self.feature_size)` as the gradient for the entire layer:

```
h_in_grad = np.zeros((batch_size, self.hidden_size))

sequence_length = x_seq_out_grad.shape[1]

x_seq_in_grad = np.zeros((batch_size, sequence_length, self.feature_si

for t in reversed(range(sequence_length)):

    x_out_grad = x_seq_out_grad[:, t, :]

    grad_out, h_in_grad = \
        self.nodes[t].backward(x_out_grad, h_in_grad, self.params)

    x_seq_in_grad[:, t, :] = grad_out
```

From this, we see that `RNNNode`s should have a `backward` method that, following the pattern, is the opposite of the `forward` method, taking in two arrays of shapes:

- `(batch_size, output_size)`
- `(batch_size, hidden_size)`

and returning two arrays of shapes:

- `(batch_size, feature_size)`
- `(batch_size, hidden_size)`

And that's the working of an `RNNLayer`. Now it seems like the only thing left is to describe the core of recurrent neural networks: the `RNNNode`s where the actual computations happen. Before we do, let's clarify the role of `RNNNode`s and their variants within RNNs as a whole.

The Essential Elements of RNNNodes

important concepts to understand about RNNs are those that we've described in the diagrams and code thus far in this chapter: the way the data is structured and the way it and the hidden states are routed between layers and through time. As it turns out, there are multiple ways we can implement `RNNNode`s, the actual processing of the data from a given time step, and updating of the layer's hidden state. One way produces what is usually thought of as a "regular" recurrent neural network, which we'll refer to here by another common term: a "vanilla RNN." However, there are other, more complicated ways that produce different variants of RNNs; one of these, for example, is a variant with `RNNNode`s called GRUs, which stands for "Gated Recurrent Units." Often, GRUs and other RNN variants are described as being significantly different from vanilla RNNs; however, it is important to understand that *all* RNN variants share the structure of the layers that we've seen so far—for example, they all pass data forward in time in the same way, updating their hidden state(s) at each time step. The only way they differ is in the internal workings of these "nodes."

To reinforce this point: if we implemented a `GRULayer` instead of an `RNNLayer`, the code would be exactly the same! The following code would still form the core of the forward pass:

```
sequence_length = x_seq_in.shape[1]

x_seq_out = np.zeros((batch_size, sequence_length, self.output_size))

for t in range(sequence_length):

    x_in = x_seq_in[:, t, :]

    y_out, H_in = self.nodes[t].forward(x_in, H_in, self.params)

    x_seq_out[:, t, :] = y_out
```

The only difference is that each "node" in `self.nodes` would be a `GRUNode` instead of an `RNNNode`. The `backward` method, similarly, would be identical.

This is also almost entirely true for the most well-known variant on vanilla RNNs: LSTMs, or "Long Short Term Memory" cells. The only difference with these is that `LSTMLayer`s require *two* quantities to be "remembered" by the layer and updated as sequence elements are passed forward through time: in addition to a "hidden state," there is a "cell state" stored in the layer that allows it to better model long-term dependencies. This leads to some minor differences in how we would implement an `LSTMLayer` as opposed to an `RNNLayer`; for example, an `LSTMLayer` would have two `ndarray`s to store the layer's state throughout the time steps:

- An `ndarray` `start_H` of shape `(1, hidden_size)`, representing the layer's hidden state
- An `ndarray` `start_C` of shape `(1, cell_size)`, representing the layer's cell state

Each `LSTMNode`, therefore, should take in the input, as well as both the hidden state *and* the cell state. On the forward pass, this will look like:

```
y_out, H_in, C_in = self.nodes[t].forward(x_in, H_in, C_in self.params
```

as well as:

```
grad_out, h_in_grad, c_in_grad = \
    self.nodes[t].backward(x_out_grad, h_in_grad, c_in_grad, self.para
```

There are many more variants than the three mentioned here, some of which, such as LSTMs with "peephole connections," have a cell state in addition to only a hidden state, and some of which maintain only a hidden state.[5] Still, a layer made up of `LSTMPeepholeConnectionNode`s would fit into an `RNNLayer` in the same way as the variants we've seen so far and would thus have the same `forward` and `backward` methods. This basic structure of RNN—the way data is routed forward through layers, as well as forward through time steps, and then routed in the opposite direction during the backward pass—is what makes recurrent neural networks unique. The actual structural differences between a vanilla RNN and an LSTM-based RNN, for example, are relatively minor, even though they can have dramatically different performance.

With that, let's look at the implementation of an `RNNNode`.

"Vanilla" RNNNodes

RNNs receive data one sequence element at a time; for example, if we are predicting the price of oil, at each time step, the RNN will receive information about the features we are using to predict the price at that time step. In addition, the RNN will have in its "hidden state" an encoding representing cumulative information about what has happened at prior time steps. We want to combine these two pieces of data—the features at the time step, and the cumulative information from all the prior time steps—into a prediction at that time step, as well as an updated hidden state.

To understand how RNNs should accomplish this, recall what happens in a regular neural network. In a feed-forward neural network, each layer receives a set of "learned features" from the prior layer, each of which is a combination of the original features that the network has "learned" is useful. The layer then multiplies these features by a weight matrix that allows the layer to learn features that are combinations of the features the layer received as input. To level set and normalize the output, respectively, we add a "bias" to these new features and feed them through an activation function.

In recurrent neural networks, we want our updated hidden state to be a combination of both the input and the old hidden state. Thus, similar to what happens in regular neural networks:

1. We first concatenate the input and the hidden state. Then we multiply this value by a weight matrix, add a bias, and feed the result through the `Tanh` activation function. This is our updated hidden state.

2. Next, we multiply this new hidden state by a weight matrix that transforms the hidden state into an output with the dimension that we want. For example, if we are using this RNN to predict a single continuous value at each time step, we'll multiply the hidden state by a weight matrix of size `(hidden_size, 1)`.

Thus, our updated hidden state will be a function of both the input received at that time step as well as the prior hidden state, and the output will be the result of feeding this updated hidden state through the operations of a fully connected layer.

Let's code this up.

RNNNODE: THE CODE

The following code implements the steps described a moment ago. Note that, just as we'll do with GRUs and LSTMs a bit later (and as we did with the simple mathematical functions we showed in Chapter 1), we save all the quantities computed on the forward pass as attributes stored in the `Node` so we can use them to compute the backward pass:

```
            params_dict: Dict[str, Dict[str, ndarray]]
            ) -> Tuple[ndarray]:
    '''
    param x: numpy array of shape (batch_size, vocab_size)
    param H_prev: numpy array of shape (batch_size, hidden_size)
    return self.x_out: numpy array of shape (batch_size, vocab_size)
    return self.H: numpy array of shape (batch_size, hidden_size)
    '''
    self.X_in = x_in
    self.H_in = H_in

    self.Z = np.column_stack((x_in, H_in))

    self.H_int = np.dot(self.Z, params_dict['W_f']['value']) \
                                + params_dict['B_f']['value']

    self.H_out = tanh(self.H_int)

    self.X_out = np.dot(self.H_out, params_dict['W_v']['value']) \
                                    + params_dict['B_v']['value']

    return self.X_out, self.H_out
```

Another note: since we're not using `ParamOperation`s here, we'll need to store the parameters differently. We'll store them in a dictionary `params_dict`, which refers to the parameters by name. Furthermore, each parameter will have two keys: `value` and `deriv`, which will store the actual parameter values and their associated gradients, respectively. Here, in the forward pass, we simply use the `value` key.

RNNNODES: THE BACKWARD PASS

The backward pass through an `RNNNode` simply computes the value of the gradients of the loss with respect to the inputs to the `RNNNode`, given gradients of the loss with respect to the outputs of the `RNNNode`. We can do this using logic similar to that which we worked out in Chapters 1 and 2: since we can represent an `RNNNode` as a series of operations, we can simply compute the derivative of each operation evaluated at its input, and successively multiply these derivatives together with the ones that have come before (taking care to handle matrix multiplication correctly) to end up with `ndarray`s representing the gradients of the loss with respect to each of the inputs. The following code accomplishes this:

```
def forward(self,
            x_in: ndarray,
            H_in: ndarray,
            params_dict: Dict[str, Dict[str, ndarray]]
            ) -> Tuple[ndarray]:
    '''
    param x: numpy array of shape (batch_size, vocab_size)
    param H_prev: numpy array of shape (batch_size, hidden_size)
    return self.x_out: numpy array of shape (batch_size, vocab_size)
    return self.H: numpy array of shape (batch_size, hidden_size)
    '''
    self.X_in = x_in
    self.H_in = H_in

    self.Z = np.column_stack((x_in, H_in))

    self.H_int = np.dot(self.Z, params_dict['W_f']['value']) \
                                + params_dict['B_f']['value']

    self.H_out = tanh(self.H_int)

    self.X_out = np.dot(self.H_out, params_dict['W_v']['value']) \
                                    + params_dict['B_v']['value']

    return self.X_out, self.H_out
```

function, and the shapes of the outputs of the `backward` function must match the shapes of the inputs to the `forward` function.

Limitations of "Vanilla" RNNNodes

Remember: the purpose of RNNs is to model dependencies in sequences of data. Thinking of modeling the price of oil as our canonical example, this means that we should be able to uncover the relationship between the sequence of features we've seen in the last several time steps and what will happen with the price of oil in the next time step. But how long should "several" be? For the price of oil, we might imagine that the relationship between what happened yesterday—one time step before—would be most important for predicting the price of oil tomorrow, with the day before being less important, and the importance generally decaying as we move backward in time.

While this is true for many real-world problems, there are domains to which we'd like to apply RNNs where we would want to learn extremely long-range dependencies. *Language modeling* is the canonical example here—that is, building a model that can predict the next character, word, or word part, given a theoretically extremely long series of past words or characters (since this is a particularly prevalent application, we'll discuss some details specific to language modeling later in this chapter). For this, vanilla RNNs are usually insufficient. Now that we've seen their details, we can understand why: at each time step, the hidden state is multiplied by *the same weight matrix* across all time steps in the layer. Consider what happens when we multiply a number by a value `x` over and over again: if `x < 1`, the number decreases exponentially to 0, and if `x > 1`, the number increases exponentially to infinity. Recurrent neural networks have the same issues: over long time horizons, because the same set of weights is multiplied by the hidden state at each time step, the gradient for these weights tends to become either extremely small or extremely large. The former is known as the *vanishing gradient problem* and the latter is known as the *exploding gradient problem*. Both make it hard to train RNNs to model the very long term dependencies (50–100 time steps) needed for high-quality language modeling. The two commonly used modifications of the vanilla RNN architectures we'll cover next both significantly mitigate this problem.

One Solution: GRUNodes

Vanilla RNNs can be described as taking the input and hidden state, combining them, and using the matrix multiplication to determine how to "weigh" the information contained in the hidden state against the information in the new input to predict the output. The insight that motivates more advanced RNN variants is that to model long-term dependencies, such as those that exist in language, *we sometimes receive information that tells us we need to "forget" or "reset" our hidden state*. A simple example is a period "." or a colon ":"—if a language model receives one of these, it knows that it should forget the characters that came before and begin modeling a new pattern in the sequence of characters.

A first, simple variant on vanilla RNNs that leverages this insight is GRUs or Gated Recurrent Units, so named because the input and the prior hidden state are passed through a series of "gates."

1. The first gate is similar to the operations that take place in vanilla RNNs: the input and hidden state are concatenated together, multiplied by a weight matrix, and then passed through a `sigmoid` operation. We can think of the output of this as the "update" gate.

2. The second gate is interpreted as a "reset" gate: the input and hidden state are concatenated, multiplied by a weight matrix, passed through a `sigmoid` operation, *and then multiplied by the prior hidden state*. This allows the network to "learn to forget" what was in the hidden state, given the particular input that was passed in.

the new hidden state.

4. Finally, the hidden state is updated to be the update gate times the "candidate" for the new hidden state, plus the old hidden state times 1 minus the update gate.

NOTE

We'll cover two advanced variants on vanilla RNNs in this chapter: GRUs and LSTMs. LSTMs are more popular and were invented long before GRUs. Nevertheless, GRUs are a simpler version of LSTMs, and more directly illustrate how the idea of "gates" can enable RNNs to "learn to reset" their hidden state given the input they receive, which is why we cover them first.

GRUNODES: A DIAGRAM

Figure 6-8 depicts `GRUNode` as a series of gates. Each gate contains the operations of a `Dense` layer: multiplication by a weight matrix, adding a bias, and feeding the result through an activation function. The activation functions used are either `sigmoid`, in which case the range of the result falls between 0 and 1, or `Tanh`, in which case the range falls between –1 and 1; the range of each intermediate `ndarray` produced next is shown under the name of the array.

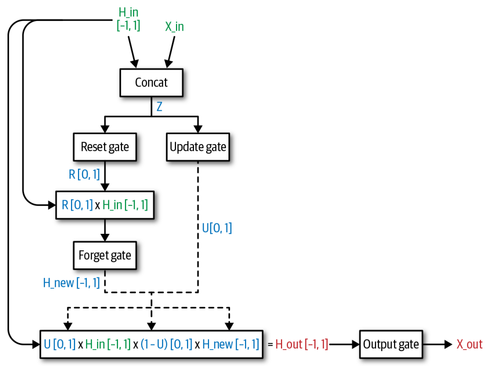

Figure 6-8. The flow of data forward through a GRUNode, passing through gates and producing X_out and H_out

In Figure 6-8 as in Figures 6-9 and 6-10, the inputs to the node are colored green, the intermediate quantities computed are colored blue, and the outputs are colored red. All the weights (not directly shown) are contained in the gates.

Note that to backpropagate through this, we would have to represent this solely as a series of `Operation`s, compute the derivative of each `Operation` with respect to its input, and multiply the results together. We don't show that explicitly here, instead showing gates (which are really groups of three operations) as a single block. Still, at this point, we know how to backpropagate through the `Operation`s that make up each gate, and the notion of "gates" is used throughout descriptions of recurrent neural networks and their variants, so we'll stick with that representation here.

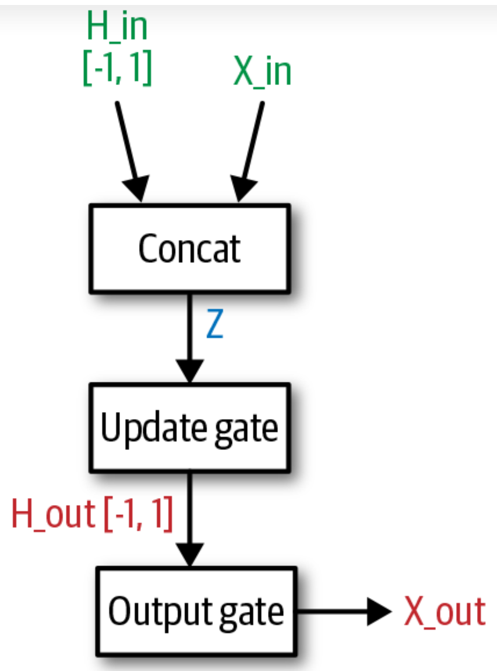

Figure 6-9. The flow of data forward through an RNNNode, passing through just two gates and producing X_out and H_out

Thus, another way to think of the `Operation`s we previously described as making up a vanilla `RNNNode` is as passing the input and hidden state through two gates.

GRUNODES: THE CODE

The following code implements the forward pass for a `GRUNode` as described earlier:

```
def forward(self,
            X_in: ndarray,
            H_in: ndarray,
            params_dict: Dict[str, Dict[str, ndarray]]) -> Tuple[ndar
    '''
    param X_in: numpy array of shape (batch_size, vocab_size)
    param H_in: numpy array of shape (batch_size, hidden_size)
    return self.X_out: numpy array of shape (batch_size, vocab_size)
    return self.H_out: numpy array of shape (batch_size, hidden_size)
    '''
    self.X_in = X_in
    self.H_in = H_in

    # reset gate
    self.X_r = np.dot(X_in, params_dict['W_xr']['value'])
    self.H_r = np.dot(H_in, params_dict['W_hr']['value'])

    # update gate
    self.X_u = np.dot(X_in, params_dict['W_xu']['value'])
    self.H_u = np.dot(H_in, params_dict['W_hu']['value'])

    # gates
    self.r_int = self.X_r + self.H_r + params_dict['B_r']['value']
    self.r = sigmoid(self.r_int)
```

```
        # new state
        self.h_reset = self.r * H_in
        self.X_h = np.dot(X_in, params_dict['W_xh']['value'])
        self.H_h = np.dot(self.h_reset, params_dict['W_hh']['value'])
        self.h_bar_int = self.X_h + self.H_h + params_dict['B_h']['value']
        self.h_bar = np.tanh(self.h_bar_int)

        self.H_out = self.u * self.H_in + (1 - self.u) * self.h_bar

        self.X_out = (
      np.dot(self.H_out, params_dict['W_v']['value']) \
      + params_dict['B_v']['value']
      )

        return self.X_out, self.H_out
```

Note that we don't explicitly concatenate `X_in` and `H_in`, since—unlike in an `RNNNode`, where they are always used together—we use them independently in `GRUNode`s; specifically, we use `H_in` independently of `X_in` in the line `self.h_reset = self.r * H_in`.

The `backward` method can be found on the book's website; it simply steps backward through the operations that make up a `GRUNode`, calculating the derivative of each operation with respect to its input and multiplying the results together.

LSTMNodes

Long Short Term Memory cells, or LSTMs, are the most popular variant of vanilla RNN cells. Part of the reason for this is that they were invented in the early days of deep learning, back in 1997,[6] whereas investigation into LSTM alternatives such as GRUs has just accelerated in the last several years (GRUs were proposed in 2014, for example).

Like GRUs, LSTMs are motivated by the desire to give the RNN the ability to "reset" or "forget" its hidden state as it receives new input. In GRUs, this is achieved by feeding the input and hidden state through a series of gates, as well as computing a "proposed" new hidden state using these gates—`self.h_bar`, computed using the gate `self.r`—and then computing the final hidden state using a weighted average of the proposed new hidden state and the old hidden state, controlled by an update gate:

```
self.H_out = self.u * self.H_in + (1 - self.u) * self.h_bar
```

LSTMs, by contrast, *use a separate "state" vector, the "cell state," to determine whether to "forget" what is in the hidden state*. They then use two other gates to control the extent to which they should reset or update *what is in the cell state*, and a fourth gate to determine the extent to which the hidden state gets updated based on the final cell state.[7]

LSTMNODES: DIAGRAM

Figure 6-10 shows a diagram of an `LSTMNode` with the operations represented as gates.

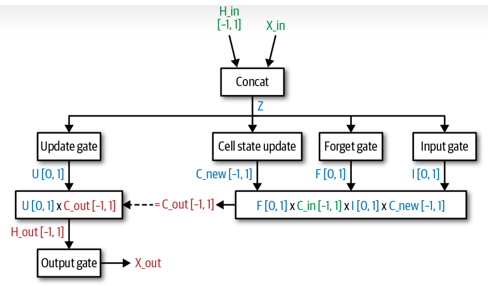

Figure 6-10. The flow of data forward through an LSTMNode, passing through a series of gates and outputting updated cell states and hidden states C_out and H_out, respectively, along with an actual output X_out

LSTMS: THE CODE

As with `GRUNode`s, the full code for an `LSTMNode`, including the `backward` method and an example showing how these nodes fit into an `LSTMLayer`, is included on the book's website. Here, we just show the `forward` method:

```
def forward(self,
  X_in: ndarray,
  H_in: ndarray,
  C_in: ndarray,
  params_dict: Dict[str, Dict[str, ndarray]]):
  '''
  param X_in: numpy array of shape (batch_size, vocab_size)
  param H_in: numpy array of shape (batch_size, hidden_size)
  param C_in: numpy array of shape (batch_size, hidden_size)
  return self.X_out: numpy array of shape (batch_size, output_size)
  return self.H: numpy array of shape (batch_size, hidden_size)
  return self.C: numpy array of shape (batch_size, hidden_size)
  '''

  self.X_in = X_in
  self.C_in = C_in

  self.Z = np.column_stack((X_in, H_in))
  self.f_int = (
    np.dot(self.Z, params_dict['W_f']['value']) \
    + params_dict['B_f']['value']
    )
  self.f = sigmoid(self.f_int)

  self.i_int = (
    np.dot(self.Z, params_dict['W_i']['value']) \
    + params_dict['B_i']['value']
    )
  self.i = sigmoid(self.i_int)

  self.C_bar_int = (
    np.dot(self.Z, params_dict['W_c']['value']) \
    + params_dict['B_c']['value']
    )
  self.C_bar = tanh(self.C_bar_int)
  self.C_out = self.f * C_in + self.i * self.C_bar

  self.o_int = (
    np.dot(self.Z, params_dict['W_o']['value']) \
    + params_dict['B_o']['value']
    )
  self.o = sigmoid(self.o_int)
  self.H_out = self.o * tanh(self.C_out)

  self.X_out = (
    np.dot(self.H_out, params_dict['W_v']['value']) \
```

And that's the last element of our RNN framework that we needed to start training models! There is one more topic we should cover: how to represent text data in a form that will allow us to feed it into our RNNs.

Data Representation for a Character-Level RNN-Based Language Model

Language modeling is one of the most common tasks RNNs are used for. How can we reshape a sequence of characters into a training dataset so that an RNN can be trained to predict the next character? The simplest method is to use *one-hot encoding*. This works as follows: first, each letter is represented as a vector with dimension equal to *the size of the vocabulary* or the number of letters in the overall corpus of text we'll train the network on (this is calculated beforehand and hardcoded as a hyperparameter in the network). Then each letter is represented as a vector with a 1 in the position representing that letter and 0s everywhere else. Finally, the vectors for each letter are simply concatenated together to get an overall representation for the sequence of letters.

Here's a simple example of how this would look with a vocabulary of four letters, `a`, `b`, `c`, and `d`, where we arbitrarily call `a` the first letter, `b` the second letter, and so on:

$$abcdb \rightarrow \begin{bmatrix} \begin{bmatrix}1\\0\\0\\0\end{bmatrix} & \begin{bmatrix}0\\1\\0\\0\end{bmatrix} & \begin{bmatrix}0\\0\\1\\0\end{bmatrix} & \begin{bmatrix}0\\0\\0\\1\end{bmatrix} & \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

This 2D array would take the place of one observation of shape `(sequence_length, num_features) = (5, 4)` in a batch of sequences. So if our text was "abcdba"—of length 6—and we wanted to feed sequences of length 5 into our array, the first sequence would be transformed into the preceding matrix, and the second sequence would be:

$$bcdba \rightarrow \begin{bmatrix} \begin{bmatrix}0\\1\\0\\0\end{bmatrix} & \begin{bmatrix}0\\0\\1\\0\end{bmatrix} & \begin{bmatrix}0\\0\\0\\1\end{bmatrix} & \begin{bmatrix}0\\1\\0\\0\end{bmatrix} & \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

These would be then concatenated together to create an input to the RNN of shape `(batch_size, sequence_length, vocab_size) = (2, 5, 4)`. Continuing in this way, we can take raw text and transform it into a batch of sequences to be fed into an RNN.

In the Chapter 6 notebook on the book's GitHub repo, we code this up as part of an `RNNTrainer` class that can take in raw text, preprocess it using the techniques described here, and feed it into an RNN in batches.

Other Language Modeling Tasks

We didn't emphasize this earlier in the chapter, but as you can see from the preceding code, all `RNNNode` variants allow an `RNNLayer` to output a different number of features than it received as input. The last step of all three nodes is to multiply the network's final hidden state by a weight matrix we access via `params_dict[W_v]`; the second dimension of this weight matrix will determine the dimensionality of the `Layer`'s output. This allows us to use the same architecture for different language modeling tasks simply by changing an `output_size` argument in each `Layer`.

For example, so far we've just considered building a language model via "next character prediction"; in this case, our output size will be equal to the size of the vocabulary: `output_size = vocab_size`. For something like sentiment analysis, however, sequences we pass in may simply have a label of "0" or "1"—

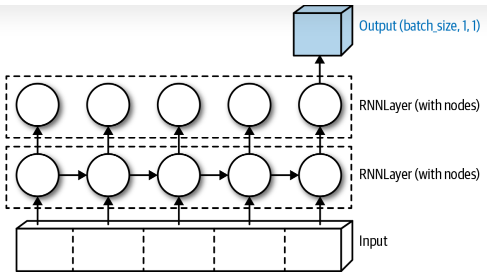

Figure 6-11. For sentiment analysis, RNNs will compare their predictions with actual values and produce gradients just for the output of the last sequence element; then backpropagation will proceed as usual, with each of the nodes that isn't the last one simply receiving an "X_grad_out" array of all zeros

Thus, this framework can easily accommodate different language modeling tasks; indeed, it can accommodate any modeling task in which the data is sequential and can be fed into the network one sequence element at a time.

Before we conclude, we'll cover an infrequently discussed aspect of RNNs: that these different kinds of layers—`GRULayer`s, `LSTMLayer`s, and other variants—can be mixed and matched.

Combining RNNLayer Variants

Stacking different kinds of `RNNLayer`s is straightforward: each RNN outputs an `ndarray` of shape `(batch_size, sequence_length, output_size)`, which can be fed into the next layer. As was the case in `Dense` layers, we don't have to specify an `input_shape`; we simply set up the weights based on the first `ndarray` the layer receives as input to be the appropriate shape given the input. An `RNNModel` can thus have a `self.layers` attribute of:

```
[RNNLayer(hidden_size=256, output_size=128),
RNNLayer(hidden_size=256, output_size=62)]
```

As with our fully connected neural networks, we just have to be sure that the last layer produces an output of the desired dimensionality; here, if we are dealing with a vocabulary of size 62 and doing next character prediction, our last layer must have an `output_size` of 62, just as the last layer in our fully connected neural networks dealing with the MNIST problem had to have dimension 10.

Something that should be clear after reading this chapter but that isn't often covered in treatments of RNNs is that, because each kind of layer we've seen has the same underlying structure taking in sequences of dimension `feature_size` and outputting sequences of dimension `output_size`, we can easily stack different kinds of layers. For example, on the book's website, we train an `RNNModel` with a `self.layers` attribute of:

```
[GRULayer(hidden_size=256, output_size=128),
LSTMLayer(hidden_size=256, output_size=62)]
```

In other words, the first layer passes its input forward through time using `GRUNode`s, and then passes an `ndarray` of shape `(batch_size, sequence_length, 128)` into the next layer, which subsequently passes them through its `LSTMNode`s.

Putting This All Together

A classic exercise to illustrate the effectiveness of RNNs is to train them to write text in a particular style; on the book's website, we have an end-to-end code example, with the model defined using the abstractions described in this chapter, that learns to write text in the style of Shakespeare. The only component we haven't shown is an `RNNTrainer` class that iterates through the training data, preprocesses it, and feeds it through the model. The main difference between this and the `Trainer` we've seen previously is that with RNNs, once we select a batch of data to be fed through—with each element of the batch simply a string—we must first preprocess it, one-hot encoding each letter and concatenating the resulting vectors into a sequence to transform each string of length `sequence_length` into an `ndarray` of shape `(sequence_length, vocab_size)`. To form the batches that get will fed into our RNN, *these* `ndarray`s will then be concatenated together to form the batch of size `(sequence_length, vocab_size, batch_size)`.

But once the data has been preprocessed and the model defined, RNNs are trained in the same way as other neural networks we've seen: batches are iteratively fed through, the model's predictions are compared to the targets to generate the loss, and the loss is backpropagated through the operations that make up the model to update the weights.

Conclusion

In this chapter, you learned about recurrent neural networks, a special kind of neural network architecture designed for processing sequences of data, rather than individual operations. You learned how RNNs are made up of layers that pass data forward in time, updating their hidden states (and their cell states, in the case of LSTMs) as they go. You saw the details of advanced RNN variants, GRUs and LSTMs, and how they pass data forward through a series of "gates" at each time step; nevertheless, you learned that these advanced variants fundamentally process the sequences of data in the same way and thus have the same structure to their layers, differing only in the specific operations they apply at each time step.

Hopefully this multifaceted topic is now less of a black box. In Chapter 7, I'll conclude the book by turning to the practical side of deep learning, showing how to implement everything we've talked about thus far using the PyTorch framework, a high-performance, automatic differentiation–based framework for building and training deep learning models. Onward!

---

1 We happened to find it convenient to arrange the observations along the rows and the features along the columns, but we didn't necessarily have to arrange the data that way. The data does, however, have to be two-dimensional.

2 Or this edition of the book, at least.

3 I want to mention an alternative solution to this problem shared by author Daniel Sabinasz on his blog, *deep ideas* (http://www.deepideas.net): he represents the operations as a graph and then uses breadth-first search to compute the gradients on the backward pass in the correct order, ultimately building a framework that mimics TensorFlow. His blog posts covering how he does this are extremely clear and well structured.

4 For a deeper dive into how to implement automatic differentiation, see *Grokking Deep Learning* by Andrew Trask (Manning).

5 See the Wikipedia page on LSTMs for more examples of LSTM variants.

6 See the original LSTM paper, "Long Short-Term Memory", by Hochreiter et al. (1997).

arranged differently.

PREV
5. Convolutional Neural Networks

NEXT
7. PyTorch