Consult the Solution Code

To take a closer look at this solution, feel free to check out the solution workspace or click here to see it as a webpage.

Complete RNN Class

I hope you tried out defining this model on your own and got it to work! Below, is how I completed this model.

> I know I want an embedding layer, a recurrent layer, and a final, linear layer with a sigmoid applied; I defined all of those in the `__init__` function, according to passed in parameters.

```
def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers, drop_prob=0.5):
        """
        Initialize the model by setting up the layers.
        """
        super(SentimentRNN, self).__init__()

        self.output_size = output_size
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim

        # embedding and LSTM layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                            dropout=drop_prob, batch_first=True)

        # dropout layer
        self.dropout = nn.Dropout(0.3)

        # linear and sigmoid layers
        self.fc = nn.Linear(hidden_dim, output_size)
        self.sig = nn.Sigmoid()
```

`__init__` explanation

First I have an **embedding layer**, which should take in the size of our vocabulary (our number of integer tokens) and produce an embedding of `embedding_dim` size. So, as this model trains, this is going to create and embedding lookup table that has as many rows as we have word integers, and as many columns as the embedding dimension.

Then, I have an **LSTM layer**, which takes in inputs of `embedding_dim` size. So, it's accepting embeddings as inputs, and producing an output and hidden state of a hidden size. I am also specifying a number of layers, and a dropout value, and finally, I'm setting `batch_first` to True because we are using DataLoaders to batch our data like that!

Then, the LSTM outputs are passed to a dropout layer and then a fully-connected, linear layer that will produce `output_size` number of outputs. And finally, I've defined a sigmoid layer to convert the output to a value between 0-1.

Feedforward behavior

Moving on to the `forward` function, which takes in an input `x` and a `hidden` state, I am going to pass an input through these layers in sequence.

```
def forward(self, x, hidden):
        """
        Perform a forward pass of our model on some input and hidden state.
        """
        batch_size = x.size(0)

        # embeddings and lstm_out
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)

        # stack up lstm outputs
        lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)

        # dropout and fully-connected layer
        out = self.dropout(lstm_out)
        out = self.fc(out)

        # sigmoid function
        sig_out = self.sig(out)

        # reshape to be batch_size first
        sig_out = sig_out.view(batch_size, -1)
        sig_out = sig_out[:, -1] # get last batch of labels

        # return last sigmoid output and hidden state
        return sig_out, hidden
```

`forward` explanation

So, first, I'm getting the `batch_size` of my input x, which I'll use for shaping my data. Then, I'm passing x through the embedding layer first, to get my embeddings as output

These embeddings are passed to my lstm layer, alongside a hidden state, and this returns an `lstm_output` and a new `hidden` state! Then I'm going to stack up the outputs of my LSTM to pass to my last linear layer.

Then I keep going, passing the reshaped `lstm_output` to a dropout layer and my linear layer, which should return a specified number of outputs that I will pass to my sigmoid activation function.

Now, I want to make sure that I'm returning *only* the **last** of these sigmoid outputs for a batch of input data, so, I'm going to shape these outputs into a shape that is `batch_size` first. Then I'm getting the last bacth by called `sig_out[:, -1], and that's going to give me the batch of last labels that I want!

Finally, I am returning that output and the hidden state produced by the LSTM layer.

`init_hidden`

That completes my forward function and then I have one more: `init_hidden` and this is just the same as you've seen before. The hidden and cell states of an LSTM are a tuple of values and each of these is size (n_layers by batch_size, by hidden_dim). I'm initializing these hidden weights to all zeros, and moving to a gpu if available.

```
def init_hidden(self, batch_size):
        ''' Initializes hidden state '''
        # Create two new tensors with sizes n_layers x batch_size x hidden_dim,
        # initialized to zero, for hidden state and cell state of LSTM
        weight = next(self.parameters()).data

        if (train_on_gpu):
            hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().cuda(),
                  weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().cuda())
        else:
            hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_(),
                      weight.new(self.n_layers, batch_size, self.hidden_dim).zero_())

        return hidden
```

After this, I'm ready to instantiate and train this model, you should see if you can decide on good hyperparameters of your own, and then check out the solution code, next!