Coding a simple RNN

This is just for encoding a class that runs a simple RNN. Other components are pre-coded and do things like:

- Tying all the parts together.
- Inputting and processing the data.
- Training the model.

First, you need to declare a class named `Emb_RNNLM(nn.Module)` like so:

`Emb_RNNLM(nn.Module):`

You need to give it this name because other parts of the code have this name to refer to it. You need it to take the argument `nn.module` so that it can use the methods in Pytorch's module for an RNN.

The indented line `def __init__(self, params):` begins the initialization of the class, passing it the argument `params` which are parameters such as the inventory size (number of phonemes plus start of string, end of string and string padding). These parameters are passed from elsewhere in the code.

The next lines pass some parameters to the model that it needs to know and define the embeddings of phonemes.

- The 'super' line enables us to automatically access the Pytorch module
- vocab size: (see above)
- embedding dimension: how many dimensions in the vectors that we use to represent phonemes. Here, the model uses 26.
- number of layers: This is a simple model with only one layer.
- hidden dimension: the embeddings, i.e. vectors that represent phonemes, are fed to a 'hidden' layer that is to contain the information of the previous phonemes that were passed to the model. Here we give it the same dimensionality as the embeddings.
- embeddings: Pytorch supplies a method for representing phonemes, words, and the like as special vectors that have a lookup table so that they can be referred to by indices.

- device: this isn't really needed but was in the original code to tell if the model is running on a cpu or a gpu (graphics processor) which can process faster, in parallel.

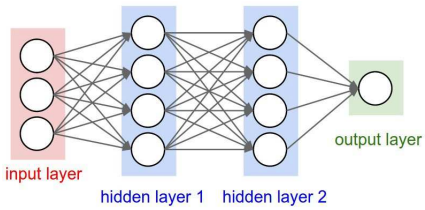
```
super(Emb_RNNLM, self).__init__()
self.vocab_size = params['inv_size']
self.d_emb = params['d_emb']
self.n_layers = params['num_layers']
self.d_hid = params['d_hid']
self.embeddings = nn.Embedding(self.vocab_size, self.d_emb)
self.device = params['device']
```

You now define the actual RNN which takes input and passes it to the recurrent layer;

```
self.i2R = nn.RNN(self.d_emb, self.d_hid,
 batch_first=True, num_layers = self.n_layers
).to(self.device)
```

- i2R stands for 'input-to-recurrent'. You could give this method any name you like, as long as you refer to it with the same name when it is called below.
- We give it the input dimension (dimension of the embeddings) and the output dimension (hidden dimension).
- The reason for 'batch first' is that even though we are not sending strings of phonemes in batches, Pytorch assumes that we want to and in other code, the vectors will be expanded to batches of vectors, so we need to tell the model how to expand them.
- As mentioned above, we have just one layer.

The output of the hidden layer is then passed through a 'linear layer'. This is a matrix of values that combines all the values of the vector that is outputted from the hidden layer according to a different set of weights for each resulting output point. The following diagram from https://towardsdatascience.com/coding-neural-network-forward-propagation-and-backpropagtion-ccf8cf369f76 shows what the network would look like of there were two hidden layers.



```
self.R2o = nn.Linear(self.d_hid, self.vocab_size).to(self.device)
```

- R2o means 'recurrent to output'.
- The linear layer takes in the output of the hidden layer and outputs a vector of the dimension of the vocabulary size.
- In training, we want the highest value of the output to be on the index of the phoneme we are trying to predict.

The next four lines refer to something discussed in Mayer and Nelson's paper, where there are advantages to having the weights of the output layer match the weights of the embeddings.

```
if params['tied']:
    if self.d_emb == self.d_hid:
        self.R2o.weight = self.embeddings.weight
    else:
        print("Dimensions don't support tied embeddings")
```

We now define in a method with Pytorch keyword 'forward' what the model actually does when it receives a phoneme as input. In our case, the 'batch' is just a single string of phonemes in a word.

```
def forward(self, batch):
```

This line takes the indices of the phonemes in the string passed to the model and converts them to the vectors that are learned as embeddings.

```
embs = self.embeddings(batch)
```

Next, those vectors are passed to the input-to-recurrent layer:

```
output, hidden = self.i2R(embs)
```

The outputs of i2R are (1) the output vectors that will be passed next to the linear layer and (2) the last hidden layer output after all the phonemes have been recurrently fed into the RNN.

The output is then fed into the linear layer:

```
outputs = self.R2o(output)
```

Both the outputs and the last hidden layer output are passed to the trainer, although it really only needs the outputs of the linear layer:

```
return outputs, hidden
```

When the whole code is run, two instances of the class defined above are created, named RNN1 and RNN2, which divide the data into two sets in the way described in the workshop.