Generate TV Scripts

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Great job you implemented everything nicely especially your work on hyperparameter tuning was commendable.
Do look at grid search or something like a TPOT which can help you automate the whole process (which would take quite a time to train though)
There are other architectures as well to try out for text generation as well as you can try to create a new dataset and tinker around applications like rap song generation or maybe write the next Game of Thrones book before the author himself.
Happy Learning :)

All Required Files and Tests

✓

The project submission contains the project notebook, called "dlnd_tv_script_generation.ipynb".

All files present

✓

All the unit tests in project have passed.

All tests passed.
Good job!

Pre-processing Data

✓

The function `create_lookup_tables` create two dictionaries:

- Dictionary to go from the words to an id, we'll call vocab_to_int
- Dictionary to go from the id to word, we'll call int_to_vocab

The function `create_lookup_tables` return these dictionaries as a tuple (vocab_to_int, int_to_vocab).

```
words = Counter(text)
 vocab = sorted(words, key = words.get, reverse = True)
 vocab_to_int = {word : ii for ii, word in enumerate(vocab)}
 int_to_vocab = {ii : word for ii, word in enumerate(vocab)}
```

Good use of counter and list comprehension inside dictionary

✓

The function `token_lookup` returns a dict that can correctly tokenizes the provided symbols.

Well defined dictionary punctuation. We are defining tokens for these punctuations so that they can be also generated and not omitted.

Batching Data

✓

The function `batch_data` breaks up word id's into the appropriate sequence lengths, such that only complete sequence lengths are constructed.

```
for i in range(0, feature_size):
       train_x[i] = words[i:i+sequence_length]
       train_y[i] = words[i+sequence_length]
```

This for loop was important to understand as we are generating sequences and the loop should jump accordingly to next value by this offset.
Good job.

In the function `batch_data`, data is converted into Tensors and formatted with TensorDataset.

```
feature_tensor = np.asarray(train_x, np.int64)
    target_tensor = np.asarray(train_y, np.int64)
    data = TensorDataset(torch.from_numpy(feature_tensor), torch.from_numpy(target_t
ensor))
    dataloader = DataLoader(data, batch_size = batch_size, shuffle = True)
```

Use of tensordataset and dataloader is done perfectly. Do read about more of the parameters offered in those two class object methods as it gives us number of workers and other multiprocessing options if the dataset is big.
https://pytorch.org/docs/stable/data.html

Finally, `batch_data` returns a DataLoader for the batched training data.

```
torch.Size([10, 5])
tensor([[ 32,  33,  34,  35,  36],
        [ 33,  34,  35,  36,  37],
        [ 15,  16,  17,  18,  19],
        [ 16,  17,  18,  19,  20],
        [ 31,  32,  33,  34,  35],
        [  0,   1,   2,   3,   4],
        [ 40,  41,  42,  43,  44],
        [ 12,  13,  14,  15,  16],
        [  4,   5,   6,   7,   8],
        [ 36,  37,  38,  39,  40]])

torch.Size([10])
tensor([ 37,  38,  20,  21,  36,   5,  45,  17,   9,  41])
```

This output is indicative of the correct implementation for batch_data.

You can avoid shuffling the training sequences if you don't need to add randomness

Build the RNN

The RNN class has complete `__init__`, `forward`, and `init_hidden` functions.

Methods are complete and all TODOs are filled up. Passes Test cases as well

✓

The RNN must include an LSTM or GRU and at least one fully-connected layer. The LSTM/GRU should be correctly initialized, where relevant.

Good job including embedding layer before lstm.
Do try Bi-LSTM as well as they give good results in text generation.
Also, you can use avoid dropout if you feel the training loss is declining faster than the validation loss after first training iteration.

RNN Training

✓

- Enough epochs to get near a minimum in the training loss, no real upper limit on this. Just need to make sure the training loss is low and not improving much with more training.
- Batch size is large enough to train efficiently, but small enough to fit the data in memory. No real "best" value here, depends on GPU memory usually.
- Embedding dimension, significantly smaller than the size of the vocabulary, if you choose to use word embeddings
- Hidden dimension (number of units in the hidden layers of the RNN) is large enough to fit the data well. Again, no real "best" value.
- n_layers (number of layers in a GRU/LSTM) is between 1-3.
- The sequence length (seq_length) here should be about the size of the length of sentences you want to look at before you generate the next word.
- The learning rate shouldn't be too large because the training algorithm won't converge. But needs to be large enough that training doesn't take forever.

```
# Sequence Length
sequence_length = 10  # of words in a sequence
# Batch Size
batch_size = 128
# Learning Rate
learning_rate = 0.001

# Model parameters
# Vocab size
vocab_size = len(int_to_vocab)
# Output size
output_size = vocab_size
```

```
# Embedding Dimension
embedding_dim = 150
# Hidden Dimension
hidden_dim = 512


# Number of RNN Layers
n_layers = 3
```

Sequence_length is ideal but do remember to check average word count per sentence in the dataset (its around 6 so anything from 6 to 10 is near resembling)
Good choice on embedding and hidden dimensions.
200 and 256 are optimal here (you can decrease the embedding size) and if you increase them the training time would significantly increase but we don't have that big of a dataset so it would probably perform the same way.

✓

The printed loss should decrease during training. The loss should reach a value lower than 3.5.

```
Epoch:   20/20    Loss: 2.952549361228943
```

Good job you pass this rubric.
You can definitely play around with n_layers and a different learning rate and train for more epochs if you want to push it down towards 2.5 or 2.

✓

There is a provided answer that justifies choices about model size, sequence length, and other parameters.

I thoroughly liked your approach toward training this model.
Your answer is very diligent and hits home almost near perfect.
Use average line word count to keep the sequence length hyperparameter optimal.

Generate TV Script

✓

The generated script can vary in length, and should look structurally similar to the TV script in the dataset.

It doesn't have to be grammatically correct or make sense.

The script is coherent quite a bit and in the show, Kramer never made sense so at least now we can make him sensible.

RETURN TO PATH