Generate TV Scripts

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Excellent work on the project! You correctly implemented and trained an LSTM-based model, with a final loss value of `3.1683` after `15` epochs, and successfully generated a script using that model.

I have included several resources in my feedback below. I would highly recommend going through some of them to see how you could extend your project beyond what you accomplished already. I would especially recommend that you check out the documentation for the Embedding Layer and try to experiment with the different parameters/attributes and observe how it impacts your output.

I hope the review helped you. If you feel there's something more that you would have preferred from this review please leave a comment. That would immensely help me to improve feedback for any future reviews I conduct including for further projects. Would appreciate your input too. Thanks!

Congratulations on finishing the project! 😄

All Required Files and Tests

✓

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

✓

All the unit tests in project have passed

All the unit tests in project have passed.

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).

```
def create_lookup_tables(text):
    """
    Create lookup tables for vocabulary
    :param text: The text of tv scripts split into words
    :return: A tuple of dicts (vocab_to_int, int_to_vocab)
    """
    # TODO: Implement Function
    unique_words=set(text)
    int_to_vocab=dict(enumerate(unique_words))
    vocab_to_int={word:idx for idx,word in int_to_vocab.items()}

    # return tuple
    return (vocab_to_int, int_to_vocab)
```

Good work! You correctly established the two dictionaries using dictionary comprehensions.

Also, here's a good resource if you wish to understand word embeddings a bit more as well. It discusses one-hot encoded and distributed vectors, and how word embeddings compare to them.

✓

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

Good job! Creating tokens for the punctuations in the input text data allows us to reduce redundancies and duplicates. For example, the words "hello" and "hello!" will no longer be considered as separate thanks to creating a token for the `!`.

If you'd like to learn more about preprocessing text data, this Kaggle Kernel is a great resource to refer to. It covers several approaches you can take as well as includes code that you could try and play around with.

Batching Data

✓

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

```
def batch_data(words, sequence_length, batch_size):
    """
    Batch the neural network data using DataLoader
    :param words: The word ids of the TV scripts
    :param sequence_length: The sequence length of each batch
    :param batch_size: The size of each batch; the number of sequences in a batch
    :return: DataLoader with batched data
    """
    # TODO: Implement function

    batch_size_total = batch_size * sequence_length
    n_batches = len(words)//batch_size_total
    words=words[:n_batches*batch_size_total]

    featute_tensor=torch.Tensor([words[i:i+sequence_length] for i in range(len(words)-se
quence_length)])
    target_tensor=torch.Tensor(words[sequence_length:])
    print(f'featute_tensor.shape, target_tensor.shape= {featute_tensor.shape,target_tens
or.shape}',end='\n\n')

    data = TensorDataset(featute_tensor, target_tensor)

    return DataLoader(data, shuffle=True, batch_size=batch_size)
```

Nice work!

You utilized a for loop (as part of a list comprehension) to create your array and fill the indices with the required values.

Do you think you could complete this step without using a for loop/list comprehension and only using numpy? Numpy is optimized to handle complex operations so that you don't usually require loops and is faster as a result. So, try it out and see if you can optimize your code or not :)

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

Great job!

For your own benefit, you could also try to create your own function which utilizes Python Generators. If you haven't worked with Generators before, I highly recommend you try them out for batching the data instead of using TensorDataset and DataLoader. Generators probably don't offer any "distinct" advantages over the two, but it's a

good python concept to be familiar with as per me.

Here's a couple of resources on generators that I found useful when I was trying to learn them -

- https://www.youtube.com/watch?v=bD05uGo_sVI
- https://jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/

The above two explain the concepts well and also have code examples that can help better grasp their functioning.

✓

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

You correctly implemented the `DataLoader`. Good job of also setting the `shuffle` parameter to `True` here. While this was explained in part in the Classroom as well, you can check out this Data Science StackExchange Post on why shuffling the data can be advantageous for machine learning/deep learning tasks.

I would also encourage you to experiment with some of the other parameters for the `DataLoader` method. You can check out the documentation for more details.

Build the RNN

✓

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

Excellent work!

I have some questions for you to think about that might benefit you -

- Does your `nn.Linear()` layer require any activation function?
- If you wanted to expand your class to be more generic such that you could have any number of layers you wanted, how would you go about that? Is that required (not just for this project, but for a general implementation which you could use anywhere for example)
- Are you familiar with Batch Normalization, yet? It's alright if not as I think you will come across this in the Classroom soon. But look it up and see how it compares to dropouts and whether they are useful for RNNs.

The reason for the above questions is to try to make sure you are ready to explore alternatives. Often, Udacity's helper code tends to tie us down a bit and we restrict ourselves from exploring further. So try to think about the above and get the answers even if they don't seem valid for this project! :)

✓

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

Nicely done!

You correctly defined -

- an embedding layer,

- an LSTM layer,
- a dropout layer, and
- a linear, fully-connected layer

Have you checked out the documentation for the Embedding Layer? Check it out and try to play around with some of the additional parameters for this method.

NOTE: In your model, you use `dropout` in `nn.LSTM` and you ALSO use it as a separate layer. In your `forward()` function, you end up with the following -

```
lstm_output, hidden=self.lstm(nn_input, hidden)
out=self.dropout(lstm_output[:,-1,:])
```

Think about how dropout is applied here. Do you get two dropout layers, one after the other? How does dropout get applied to `nn.LSTM`? Look up the documentation, if you'd like to, and see whether you actually need more dropout layers or not.

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.**

You selected a good set of hyperparameters.

- You get a low training loss with the number of epochs, which is good. We should tend to select the number of epochs such that you get a low training loss that reaches kind of a steady-state (not much change in value beyond a point).
- You selected a good batch size. Try to observe what happens if you lower it or increase it. Smaller batch sizes take too long to train. Larger batch sizes speed up the training but can degrade the quality of the model. Here is a useful resource to understand this better - Trade-off between batch size and number of iterations to train a neural network
- Your current RNN size (number of layers) fits the data well. What results do you think you will get when you increase or decrease that? First try increasing and see if your model converges better and has a lower loss or not. Changing this is limited to your system configuration as well, so you might not be able to have a very high value either.
- Usually, we select the sequence length so that it matches the structure of data we are working with. There

isn't really a rule of thumb though -Feasible sequence length for an RNN to model. For a tougher challenge, try to think of having variable sequence lengths. Variable-sized mini-batches might help in this regard.

A couple of good things for you to try out - a slightly smaller learning rate with more epochs. As you can see your loss seems to still decrease, which means your model seems to still be learning so you can train for more epochs. But at the same time your learning rate results in the loss being a bit spiky (it increases and decreases a bit too frequently instead of a steady decrease). That usually means that a higher learning rate has been selected. So you can try to reduce that as well to have the training be more stable.

Good work!

✓

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

Excellent job on getting a final loss value of ~ `3.1683` ! You can still experiment more with your hyperparameters, if you'd like to. But you are getting good results.

✓

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

> ```
> Yes, I tried different sequence lengths under 10 but 10 was good
> a number of 2 hidden layers was my go to and it did meet my expectations
> I tried hidden dims of 128,256,512 and 512 was the best
> I first chose a training epochs of 10 that did well and increased them to 15 to s
> ee how the model will continue learning. it became a little bit better and we can
> continue training for a higher number of epochs
> ```

Nice work with the experimentation!

However, one small advice. You are currently focusing on tuning the parameters to get the desired result which is definitely a good thing because a lot of it is indeed experimentation. But, try to also develop an intuitive understanding of what each parameter does and tune them accordingly. Each hyperparameter corresponds to some aspect of the model and it's much more helpful to understand what it does. Check my suggestions and links for the different hyperparameters in one of the previous rubric comments for this as well.

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.

Your generated script looks awesome!

Your generated script looks awesome!

Question for you to think about - How do you think you could design your model so that you can control how many words per dialogue your script has? Do you think that's possible with just deep learning?

DOWNLOAD PROJECT

RETURN TO PATH