UDACITY

‹ Return to Classroom DISCUSS ON STUDENT HUB

Generate TV Scripts

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Good job you implemented everything nicely.

- Remember to always not introduce dropout early in your training process and only add it if needed.
- Do look at grid search or something like a TPOT or Pycaret which can help you automate the whole process of finding best hyperparameters (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.
- Do try to read about these interesting approaches and compare and contrast which ones work the best if you ever end up having a use case and you can find an implemented paper here -> https://paperswithcode.com/task/text-generation

All the best for your ML/AI journey :)

All Required Files and Tests

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

All required files present.

All the unit tests in project have passed.

All tests passed. Quality work!

- You can definitely use more assert statements instead of print statements to check batch sizes and debug a few more test cases as we have quite a lot of preprocessing/transformation in batching data.
- Also, use logging for different warnings/errors so that with version deprecation of functions, it is easy to find the culprit breaking the code within the logs as package versions keep on updating a lot.
- Proper variable naming, ample whitespaces, consistent indentation, and breaking up the code into blocks, maximum code reuse, and refactoring code into functions or lambdas aids readability. A very succinct guide can be found in this StackExchange answer.

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).

```
    # TODO: Implement Function
    word_counts = Counter(text)
    sorted_vocab = sorted(word_counts, key=word_counts.get, reverse=True)
    int_to_vocab = {ii: word for ii,word in enumerate(sorted_vocab)}
    vocab_to_int = {word: ii for ii, word in int_to_vocab.items()}
    # return tuple
    return (vocab_to_int , int_to_vocab)
```

- Good use of Counter. You can alternatively use set data class as well here.
- Resource for more Counter class examples -> https://www.guru99.com/python-counter-collections-example.html
- Resource for more Set data structure examples -> https://www.programiz.com/python-programming/set

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

```
-   Good use of Set. You can alternatively use counter data class as well here.
-   Resource for more Counter class examples -> <https://www.guru99.com/python-co
unter-collections-example.html>
```

```
-   Resource for more Set data structure examples -> <https://www.programiz.com/p
ython-programming/set>
```

- Well defined dictionary `token_dic` returned properly.

- We are defining tokens for these punctuations so that they can be also generated and not omitted.\
  Best way to deal with punctuations

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 ii in range(0, len(words)- sequence_length):
         x.append(words[ii:ii+sequence_length])
         y.append(words[ii + 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.
- Use of list comprehension can be done here as it makes this code more idiomatic here as well but you should always be careful while using them
  When to use list comprehensions

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

```
    #convert numpy arrays to tensors
    x_tensors = torch.from_numpy(np.array(x))
    y_tensors = torch.from_numpy(np.array(y))



    #Dataset wrapping tensors
    data = TensorDataset(x_tensors, y_tensors)

    #multi-process iterators over the dataset (our data loader)
    data_loader = torch.utils.data.DataLoader(data, shuffle=True,
                                              batch_size=batch_size)
```

- 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.
- You can also use num_workers = -1 to utilize all CPUs present in a device if you want to utilise

maximum possible parallelization.
https://pytorch.org/docs/stable/data.html

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

```
torch.Size([10, 5])
tensor([[  9,  10,  11,  12,  13],
        [ 22,  23,  24,  25,  26],
        [  4,   5,   6,   7,   8],
        [ 16,  17,  18,  19,  20],
        [  5,   6,   7,   8,   9],
        [ 31,  32,  33,  34,  35],
        [  0,   1,   2,   3,   4],
        [ 38,  39,  40,  41,  42],
        [ 32,  33,  34,  35,  36],
        [ 18,  19,  20,  21,  22]])

torch.Size([10])
tensor([ 14,  27,   9,  21,  10,  36,   5,  43,  37,  23])
```

You can avoid shuffling the training sequences and check if it makes any difference.

Should you shuffle?

Build the RNN

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

```
        self.vocab_size = vocab_size
        self.output_size = output_size
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.dropout = nn.Dropout(0.25)
        # define model layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                            dropout = dropout, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_size)
```

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.

```
        batch_size = nn_input.size(0)
        nn_input = nn_input.long()
        embeds = self.embedding(nn_input)
        lstm_out , hidden = self.lstm(embeds,hidden)
        lstm_out = lstm_out.contiguous().view(-1,self.hidden_dim)
        output = self.dropout(lstm_out)
        output = self.fc(output)
        output = output.view(batch_size, -1, self.output_size)
        out = output[:, -1]

        # return one batch of output word scores and the hidden state
        return out, hidden
```

- Good job including embedding layer before lstm.
- Do try Bi-LSTM as well as they give good results in text generation. Text generation with Bi-LSTM
- Also, you can use less internal dropout (default 0.5 is a bit too much) if you feel the training loss is not decreasing after 10-15 epochs as here our aim is to overfit on the dataset to generate somewhat replica of it. You can also remove the additional dropout you added.

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 = 256
```

```
_
```

- The sequence length is ideal.
- Do always check the average word count per sentence in the dataset and here it was around 6 so anything from 6 to 10 is near resembling.

```
# Number of Epochs
num_epochs = 10
# Learning Rate
learning_rate = 0.001
```

- Number of epochs is decent. You can get loss of 3.5 or less in just 10 or less epochs its just dropout which is causing slow convergence to the loss values.
- Learning rate is good for this project. Use 0.001- 0.005 in general as a starting point but if you feel loss is not decreasing fast enough then increase it.

```
vocab_size = len(vocab_to_int)
# Output size
output_size = vocab_size
# Embedding Dimension
embedding_dim = 200
# Hidden Dimension
hidden_dim = 300
# Number of RNN Layers
n_layers = 2
```

- Good choice on embedding and hidden dimensions although when they are high, they would take quite a lot of time to train and as well.
- 200 and 256 are optimal here if you want to balance the time taken vs loss requirement here and if you increase any one of 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 and not make more difference in text generation
- There is this framework called Optuna -> https://optuna.org/
  You can use it to find optimal hyperparameters as well.

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

```
Epoch: 10/10 Loss: 3.501194214820862
```

- Just breaching the rubric requirement. It is still a bit above it but is acceptable.
- 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 even below 1.
- It doesn't necessarily mean that the generation will be better as it is still human perspective which has to judge that but pushing for lower loss would be good.
- You can look at BLEU score if you really want to understand the accuracy of generated models.

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

```
Answer: (Write answer, here) Actually larger sequence lengths were taking a long
time in training but not improving the loss value considerably, so just went with
10 as an average length of common sentences in tv scripts and alike. hidden_dim a
nd n_layers can be most easily decided by using some standard architectures found
in the literature, if you don't find a state-of-the-art architecture in the liter
ature then you must go with an experimentation strategy and should stop increasin
g the layers and number of units in each layer once the increase doesn't increase
or rather decrease the performance of your neural network. I Google searched some
networks and found the given layers and units to be performing well so chose the
m.
```

If possible whenever analysing the impact of hyperparameters, try to structure your experiments into a table for each of them which would be better for presenting your observations and findings in general for any project you work on while modelling NN based algorithms.

Some tips -

- Always use average line word count to keep the sequence length hyperparameter optimal. If you take up longer sequences then you will be generating/overfitting on bigger story chunks/dialogues which may align with your usecase or not.
- Also for embedding size, some people also use the 4th root of vocab size (categories) which would be around 20 here in our case so you can try that logic as well https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html
- You can also track the time it takes for each of your experiments for different parameters and compare and contrast the loss values vs time take for each experiment to find the perfect balance.

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.

```
george:(pointing) oh yeah, yeah, i know.

elaine: i don't know..

kramer: well, i think i could get back.

jerry: oh, you got the job?
```

```
kramer: yeah, i don't care.(george enters.)
```

The script is coherent quite a bit and in the show,George was always sarcastic getting into situations with his words (and vandelay was his alter ego) while Kramer never made sense so at least now we can make them

better...or can we? :P

https://www.youtube.com/watch?v=JtA8gqWA6PE

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review

START