UDACITY

‹ Return to Classroom

DISCUSS ON STUDENT HUB

Generate TV Scripts

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Dear Student,

Well done ⭐ you have completed the tv script generation project with flying colours.

I really like that you have achieved the required threshold value(3.5) for the training loss in 7 epochs.

```
Epoch: 7/10 Loss: 3.4895895872116087
```

I hope you also know that the current generated script doesn't make sense for now. if you add more real script data and trained with a big model of LSTM then the model will be generic.

Best of luck 👍 for the next nano degree assignments. till that time, happy learning 📚

All Required Files and Tests

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

Good 👏 the notebook is available in the submission.
✅ notebook file

All the unit tests in project have passed.

superb all unit tests are passing 🔥
✅ create_lookup_tables functionality implementation.
✅ token_lookup functionality implementation.

✅ RNN model implementation.
✅ forward and back propagation for RNN model.

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).

Good 👌 The implementation is very basic python programming.

But, to implement an optimized program, you can use counter and dictionary comprehension as follow:

```
from collections import Counter
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
    vocabs = Counter(text)

    vocabs_sorted = sorted(vocabs, key=vocabs.get, reverse = False)
    vocab_to_int = {word: i for i, word in enumerate(vocabs_sorted)}
    int_to_vocab = {i: word for i, word in enumerate(vocabs_sorted)}

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

in the future, you can also use predefined word embedding for better results.
please refer to the link for more about word embedding

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

great 👍 all tokens are considered in the function.

- Period (.)
- Comma (,)
- Quotation Mark (")
- Semicolon (;)
- Exclamation mark (!)
- Question mark (?)
- Left Parentheses (( )
- Right Parentheses ( ))
- Dash (-)
- Return (\n)

Batching Data

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

good 👏
Here, instead of using separated lists, you can directly use NumPy array with comprehensive lists to converting datatype from list to array and array to tensor in an optimized way.

```
def batch_data(words, sequence_length, batch_size):

    features = np.array([words[idx:idx+sequence_length] for idx, word in e
numerate(words[0:-sequence_length])])
    targets = np.array([words[idx+sequence_length] for idx, word in enumer
ate(words[0:-sequence_length])])
    data = TensorDataset(torch.from_numpy(features), torch.from_numpy(targ
ets))
    data_loader = DataLoader(data,batch_size=batch_size)

    return data_loader
```

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

Nice 👌 you have used `TensorDataset` to convert tensors to TensorDataset format.

```
    data = TensorDataset(torch.from_numpy(np.asarray(x)), torch.from_numpy
(np.array(y)))
```

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

Nice 👌 you have used the internal PyTorch function to create batch data.

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

Build the RNN

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

Good implementation 👏 you have initialized both the embedding layer and LSTM cells layer perfectly.

```
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers, dropout=d
ropout, batch_first=True)
```

good, the stack up LSTM output is implemented correctly.

```
        lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)
```

for more explanation about suitable architecture for text generation here

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 that you have chosen the LSTM model over the GRU model.

There are multiple benefits of LSTM and GRU:

- GRU couples forget as well as input gates. GRU uses less training parameters and therefore use less memory, execute faster and train faster than LSTM
- LSTM is more accurate on the dataset using longer sequences.

In short, if the sequence is large or accuracy is very critical, please go for LSTM whereas for less memory consumption and the faster operation go for GRU
click here to understand difference between LSTM and GRU.

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.

all hyperparameters seem good.
you can also increase your training performance by increasing word embedding and sequence length.
you can also refer to the link to improve the hyperparameters

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

perfect 🔥 the loss value is lower than 3.5

```
Epoch: 10/10 Loss: 3.396915725708008
```

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

good 👏 efforts to tune hyper-parameters with multiple set by changing,

- sequence_length
- batch_size
- num_epochs
- learning_rate
- embedding_dim
- hidden_dim
- n_layers

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 generated script looks good 🔥

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review

START