UDACITY

‹ Return to Classroom

Generate TV Scripts

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Congratulations 🎉🎉🎉

- Your submission reveals that you have made an **excellent effort** in finishing this project,especially the batching, model architecture and hyperparameters.
- Very good hyperparameters and decreasing cross entropy loss. It is great that you have got everything right in first review 👍
- Please go through the additional suggestions in the rubric below.
- I wish you all the best for next adventures 🚀

Few references to explore:

- Colah's Blog A visual explanation of LSTMs you want to look at to understand it more.
- Andrej Karpathy Andrej is director of AI and AutoPilot Vision at Tesla, this link explains the unreasonable effectiveness of RNN
- Rohan Kapur , This link is an overall explanation of RNNs that you may find useful and insightful.

All Required Files and Tests

The project submission contains the project notebook, called “dlnd_tv_script_generation.ipynb”.

All required files are present 👍

Bonus tips:

- It is recommended to export your conda environment into environment.yaml file using command **conda env export -f environment.yaml**, so that you can recreate your conda environment later.
- While submitting this to any version control system like Github, make sure to include helper, data and environment files and exclude and temp files. It will help you in future if you want to re-execute it. Some guideline for best practice.

All the unit tests in project have passed.

All the unit tests in project have passed. 👍

Donald Knuth (a famous computer science pioneer) once famously said about unit tests:
"Beware of bugs in the above code; I have only proved it correct, not tried it."

Article on unit-test in machine learning system here

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 job! 👏

- The Counter is also an convenient way to get the information needed for that approach

```
from collections import Counter

 word_counts = Counter(text)
    sorted_words = sorted(word_counts, key=word_counts.get, reverse=True)
    vocab_to_int = dict()
    int_to_vocab = dict()

    for i, word in enumerate(sorted_words):
        vocab_to_int[word] = i
        int_to_vocab[i] = word
    return vocab_to_int, int_to_vocab
```

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

👍

- Converting each punctuation into explicit token is very handy when working with RNNs.
- All 10 entries are present
- Do read up this link to understand what other pre-processing steps are carried out before feeding text data to RNNs.
- You can keep it line-separeted to make it more readable :

```
return {
      '.'  : '||period||',
      ','  : '||comma||',
      '"'  : '||quotationmark||',
      ';'  : '||semicolon||',
      '!'  : '||exclamationmark||',
      '?'  : '||questionmark||',
      '('  : '||leftparentheses',
      ')'  : '||rightparentheses',
      '--' : '||doubledash||',
      '\n' : '||return||'
 }
```

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 Job 👏

- The implementation breaking up word id's into the appropriate sequence lengths

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

It is recommended to add explanatory comments in between, look at this alternative implementation :

```
    # get number of targets we can make
    n_targets = len(words) - sequence_length
    # initialize feature and target
    feature, target = [], []
    # loop through all targets we can make
    for i in range(n_targets):
        x = words[i : i+sequence_length]    # get some words from the given list
```

```
            y = words[i+sequence_length]        # get the next word to be the target
            feature.append(x)
            target.append(y)

        feature_tensor, target_tensor = torch.from_numpy(np.array(feature)), torch.fr
om_numpy(np.array(target))
        # create data
        data = TensorDataset(feature_tensor, target_tensor)
        # create dataloader
        dataloader = DataLoader(data, batch_size=batch_size, shuffle=True)
         # return a dataloader
        return dataloader
```

This implementation is basically responsible for loading sequenced data into Tensors in order for PyTorch's TensorDataset utility to generate the dataset.

```
data = TensorDataset(feature_tensors, target_tensors)
```

Check this dataloading tutorial

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

The unit test output tensor verifies the implementation 👍

- The function of Dataloader is to combine a dataset and a sampler, and finally provides an iterable over the given dataset. Feature like automatic batching are also supported. Check details of Dataloader here
- Adding `shuffle=True` in the dataloader is allowing you to add randomness in the training sequences.

Build the RNN

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

👍

- `__init__`, `forward` and `init_hidden` functions are complete, a good model architecture
- RNN implements an LSTM Layer, and initializes it appropriately.

```
self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                          dropout=dropout, batch_first=True)
```

- we can also use GRU as well in place of LSTM:

```
self.embedding = nn.Embedding(self.vocab_size,self.embedding_dim)
self.GRU = nn.GRU(self.embedding_dim,self.hidden_dim,self.n_layers,batch_fi
rst=True,dropout=self.dropout)
self.fc = nn.Linear(self.hidden_dim,self.output_size)
```

- Check this article: Difference between LSTM and GRU

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

The ideal structure is as follows:

- Embedding layer (nn.Embedding) before the LSTM or GRU layer.
- The fully-connected layer comes at the end to get our desired number of outputs.
- As implemented, It is also **recommended to not use a dropout after LSTM and before FC layer**, as the drop out is already incorporated in the LSTMs, A lot of students adds it and then end up finding convergence difficult.

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.**

🚀

- Enough epochs to get near a minimum in the training loss.
- Batch size is large enough to train efficiently. In order to use the GPU more efficiently, we can always try to set a value that is a power of two (e.g. 64 or 128 or 256)
- Sequence length is about the size of the length of sentences we want to generate. Considering the fact that there are approximately an average of 11.504 words per line and 15.248 sentences in each scene
- Size of embedding is in the range of [200-300]. The vocab contained ~46,367 unique words. Now you can try to cut this down significantly by 98% to 1000 embeddings. For example, Google's news word

vectors, the GloVe vectors, and other word vectors are usually in the range 50 to 300
- Learning rate seems good based on other hyper parameter
- Hidden Dimension: 128-256 hidden dimensions to give the network a solid amount of features/states to learn from. recommendation on how to select Hidden Dimension
  `hidden_dim = 5*int(len(train_loader.dataset) / (embedding_dim + output_size))`
- Number of layers is in between 1-3 as suggested in the project.

> Your efforts shows that you have really have thought about it to get an optimized value 🔥

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

🏁 excellent decreasing loss ..

```
Epoch:   17/20    Loss: 3.2620907768726894

Epoch:   18/20    Loss: 3.238267054377139

Epoch:   19/20    Loss: 3.21063452883785

Epoch:   20/20    Loss: 3.1862994286003943

Model Trained and Saved
```

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

detailed point-wise answer discussing approach and reasoning behind the answer.

> The act of elaborating your approach often leads to a deeper understanding of the material 😊

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.

well generated fun script! 👏

- all the lines are making sense
- sentences are grammatically intact

DOWNLOAD PROJECT

RETURN TO PATH