UDACITY

‹ Return to "Deep Learning" in the classroom

Generate TV Scripts

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Kudos ! I think you've done a perfect job of implementing a recurrent neural net fully. It's very clear that you have a good understanding of the basics. Keep improving and keep learning.

Further, I would highly suggest you implement RNN from scratch in Python/Numpy. It would be a very good exercise and will lead to better understanding of abstraction provided by PyTorch. This gist can be a good starting point (along with this video).

All Required Files and Tests

✓

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

The `ipython` notebook and helper files are included.

✓

All the unit tests in project have passed.

Great work. Unit testing is one of the most reliable methods to ensure that your code is free from all bugs without getting confused with the interactions with all the other code. If you are interested, you can read up more and I hope that you will continue to use unit testing in every module that you write to keep it clean and speed up your development.
But always keep in mind, that unit tests cannot catch every issue in the code. So your code could have bugs even though unit tests pass.

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).

Clean and concise.
Mapping each char to unique identifier (int) and vice-versa is always a good approach when working with text data.
Further, when generating new text, this will be of utmost importance.

✓

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

Perfect, as required.
Converting each punctuation into explicit token is very handy when working with RNNs.

Pro Tip: Do read up this link to understand what other pre-processing steps are carried out before feeding text data to RNNs.

Batching Data

✓

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

Implemented with perfection ! 👍

Further, I highly encourage you to write unit test for this function. A sample unit test can be constructed as below.

```
def test_batch_data(lst, seq_len, batch_size, expected_nb_batches, expected_nb_ex
amples):
    nb_batches = 0
    nb_examples = 0
    dl = batch_data(lst, seq_len, batch_size)
    for x, y in dl:
        print(x.shape)
        nb_batches += 1
        nb_examples += x.size(0)
        assert x.size() == (batch_size, seq_len), " x.size(): {} found, expected
 {}".format(list(x.size()), [batch_size, seq_len])
```

```
{} .format(list(x.size()), [batch_size, seq_len])
        assert y.size() == (batch_size,), "y.size(): {} found, expected {}".forma
t(y.size(), (batch_size,))

    assert expected_nb_batches == nb_batches, "nb_batches: {}, expected {}".forma

t(nb_batches, expected_nb_batches)
    assert expected_nb_examples == nb_examples, "nb_examples: {}, expected {}".fo
rmat(nb_examples, expected_nb_examples)
    print("Done!")
```

```
test_batch_data(list(range(0, 20)), 6, 4, expected_nb_batches=3, expected_nb_exam
ples=12)
test_batch_data(list(range(0, 20)), 4, 5, expected_nb_batches=3, expected_nb_exam
ples=15)
test_batch_data(list(range(0, 10)), 3, 3, expected_nb_batches=2, expected_nb_exam
ples=6)
```

- If there is still some doublt, this video will defitanely help you better understand how batching operates.

✓

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

✓

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

Pro Tip: PyTorch has an amazing official library for text based Data loaders and abstractions in NLP called torchtext. This is an awesome package, for example check out this Comprehensive Introduction to Torchtext.

Build the RNN

✓

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

✓

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

RNN Class, the hidden state and Embeddings looks all good to me.

When working with RNN, the hidden state is what preserves the context of the text. Often behind the math of LSTM/GRU, the intuition behind the hidden state is lost.

One hand-wavy analogy of the hidden state is, it corresponds to the **thoughts** in our brain. With each passing time step, we think new thought (hidden state) based on previous thought (previous state). This thinking may or may not lead to action (output vector).

Further Resource: Keras has a neat API for visualizing the architecture, which is very helpful while debugging your network. `pytorch-summary` is a similar project in PyTorch.

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.**

15 `epochs` with around 0.0005 `learning_rate` took you places. The hyperparams chosen are very good as evident from your training loss. Further, setting `batch_size` as a power of 2 (128 in your case) is handled efficiently by PyTorch (better so on GPU).

Everything worked perfectly with these setting of hyperparams. Seems to me, you had your **Deep learning hat** on while choosing these. 😉

✓

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

✓

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

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.

Woah ! Now that is something 👍

Being a Seinfeld fan myself, these conversations are amazing knowing they are produced by an RNN. I am sure training on the whole series will produce better results, who knows, an episode itself.

Fun Fact: Google is using RNN to generate art (music, sketches). The project is named Magenta. You can even play with a demo in browser.

DOWNLOAD PROJECT

RETURN TO PATH