Generate TV Scripts

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Congratulation!!, 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. 👍

Required Files and Tests

✓

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

The `dlnd_tv_script_generation.ipynb` , `helper.py` , `problem_unittests.py` and `HTML` 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. 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.

Preprocessing

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 in the 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.

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

Build the Neural Network

Implemented the `get_inputs` function to create TF Placeholders for the Neural Network with the following placeholders:

- Input text placeholder named "input" using the TF Placeholder name parameter.
- Targets placeholder
- Learning Rate placeholder

The `get_inputs` function return the placeholders in the following the tuple (Input, Targets, LearingRate)

All correct !

The `get_init_cell` function does the following:

- Stacks one or more BasicLSTMCells in a MultiRNNCell using the RNN size `rnn_size`.
- Initializes Cell State using the MultiRNNCell's `zero_state` function
- The name "initial_state" is applied to the initial state.
- The `get_init_cell` function return the cell and initial state in the following tuple (Cell, InitialState)

RNN Cell and the hidden state looks all good to me.
Dropout layers have been added in order to reduce network overfitting.

Dropout is a regularization method where input and recurrent connections to LSTM units are probabilistically excluded from activation and weight updates while training a network. This has the effect of reducing overfitting and improving model performance.

The function `get_embed` applies embedding to `input_data` and returns embedded sequence.

Correct use of `tf.nn.embedding_lookup`. Embedding are necessary because Neural Nets need numbers to crunch instead of words.

Although you used `tf.random_normal` distribution, there are other ways to create these embeddings. Using a `tf.truncated_normal` distribution, with a small standard deviation can be very good.

The function `build_rnn` does the following:

- Builds the RNN using the `tf.nn.dynamic_rnn`.
- Applies the name "final_state" to the final state.
- Returns the outputs and final_state state in the following tuple (Outputs, FinalState)

`tf.nn.dynamic_rnn` used correctly.

The `build_nn` function does the following in order:

- Apply embedding to `input_data` using `get_embed` function.
- Build RNN using cell using `build_rnn` function.
- Apply a fully connected layer with a linear activation and `vocab_size` as the number of outputs.
- Return the logits and final state in the following tuple (Logits, FinalState)

Congrats on placing all the puzzles together ! You correctly built your RNN :)

One point of note is, tensorflow abstracts a lot of detailed theory behind RNN/LSTM. In the long run, you would definitely want to understand these concepts by heart. C Olah and Andrej are two researchers who explains these concepts wonderfully.

The `get_batches` function create batches of input and targets using `int_text`. The batches should be a Numpy array of tuples. Each tuple is (batch of input, batch of target).

- The first element in the tuple is a single batch of input with the shape [batch size, sequence length]
- The second element in the tuple is a single batch of targets with the shape [batch size, sequence length]

A good implementation.

A better implementation, making use of numpy vector operations can be:

```python
def get_batches(int_text, batch_size, seq_length):

    n_batches = int(len(int_text) / (batch_size * seq_length))

    # Drop the last few characters to make only full batches
    xdata = np.array(int_text[: n_batches * batch_size * seq_length])
    ydata = np.array(int_text[1: n_batches * batch_size * seq_length + 1])

    x_batches = np.split(xdata.reshape(batch_size, -1), n_batches, 1)
    y_batches = np.split(ydata.reshape(batch_size, -1), n_batches, 1)

    return np.array(list(zip(x_batches, y_batches)))
```

Neural Network 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.**
- **Size of the RNN cells (number of units in the hidden layers) is large enough to fit the data well. Again, no real "best" value.**
- **The sequence length (seq_length) here should be about the size of the length of sentences you want to generate. Should match the structure of the data.**
  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.
  Set show_every_n_batches to the number of batches the neural network should print progress.

1000 `epochs` with around 0.001 `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 (1000 in your case) is handled efficiently by tensorflow (better so on GPU).

However, the `epochs` and `batch_size` are too high which would lead to overfitting problem, you may try training it later with lower `epochs` and `batch_size`.

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

✓

The project gets a loss less than 1.0

Good job!!

Generate TV Script

"input:0", "initial_state:0", "final_state:0", and "probs:0" are all returned by `get_tensor_by_name`, in that order, and in a tuple

Correctly implemented.

The `pick_word` function predicts the next word correctly.

Use of slight bit of randomness is helpful when predicting the next word. Otherwise, the predictions might fall into a loop of the same words.

A simpler implementation using randomness will be:

```
def pick_word(probabilities, int_to_vocab):

    word_idx = np.random.choice(len(probabilities), size=1, p=probabilities)[0]
    pred_word = int_to_vocab[word_idx]
    return pred_word
```

The generated script looks similar to the TV script in the dataset.

It doesn't have to be grammatically correct or make sense.

Finally!, the generated script is similar to the script in the dataset.
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.

DOWNLOAD PROJECT

RETURN TO PATH