UDACITY

Back to Deep Learning Nanodegree

Generate TV Scripts

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Congratulations 🎉

- Your submission reveals that you have made a great effort in finishing this project. It is an important milestone in learning about RNNs
- Very good hyperparameters and loss . It is great that you have got everything right in first review 👍
- I wish you all the best for next adventures 🚀
- Nice Read: (Colah's Blog) http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- Nice Read: (Andrej Karpathy) : http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Keep up the good work 👍 Stay Udacious

Required Files and Tests

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

All required files are present 👍

- I suggest you to export your conda environment into environment.yaml file. command **conda env export -f environment.yaml**

All the unit tests in project have passed.

Well Done ! 👍 Donald Knuth (a famous computer science pioneer) once famously said

> "Beware of bugs in the above code; I have only proved it correct, not tried it."

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)

Good work 👍

- In particular, using a python set function to ensure each entry in the vocab list is unique is an excellent technique to remove duplicates from a list, and something that you will find yourself doing as part of almost any dataset-preparation work.
- when creating lookup tables it can also be helpful to index words by the frequency each word occurs in the text. The python Counter function (part of the collections library) is a convenient way to get the information needed for that approach.
  https://pymotw.com/2/collections/counter.html

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.
- Do read up this link to understand what other pre-processing steps are carried out before feeding text data to RNNs.
- An alternate implementation would be :

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

```
        '\n' : '||return||'
    }
```

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)

👏

- It is recommended to run codes using GPUs
- I like the fact that you named each tf tensor, and not just the one required by the rubric. This is very helpful in debugging, and something I always do for every tf placeholder.
- You have used named-parameter-passing for all variables where the meaning isn't obvious. For example, using `shape=[None, None]` when passing the default shape to the placeholder function.

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)

Awesome !

- Good work with dropoutwrapper too !

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

Good Work 👍

- Alternatively, you could also use `return tf.contrib.layers.embed_sequence(input_data, vocab_size, embed_dim)`
- Check this video on embedding

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)

- implemented correctly
- good practice to add comments

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)

- When activation function is specified as 'None', it takes linear activation
- Reference on activation https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0
- You could also use **tf.layers.dense** here, It is essentially the same thing https://stackoverflow.com/questions/44912297/are-tf-layers-dense-and-tf-contrib-layers-fully-connected-interchangeable11

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]

Awesome !

- You implementation reveals that you understand batching really well.
- Good use of numpy
- I suggest you to add some comments to describe it. It is good practice
- I would like to have seen a debug statement showing that

```
print(get_batches([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20], 3, 2) )
```

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.

🚀

- Enough epochs to get near a minimum in the training loss. 👍
- Batch size is large enough to train efficiently 👍
- Size of the RNN cells is large enough to fit the data well 👍
- Sequence length is about the size of the length of sentences we want to generate 👍
- Size of embedding is in the range of [200-300] 👍
- Learning rate seems good based on other hyper parameter 👍

> Your efforts shows that you have really have executed it again and again to get an optimized value 🔥

The project gets a loss less than 1.0

🏁 excellent

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

The `pick_word` function predicts the next word correctly.

- This works, but you should add in some randomness that takes in probabilities as a parameter instead of always choosing the highest probability word.

- Your current implementation will always choose the highest probability word, and it is preferable so the predictions don't fall into a loop of the same words (or sequences).

```
vocab_list = ["hello", "how", "is"]
probabilities = [0.3,0.5,0.2]
np.random.choice(vocab_list, p=probabilities)
```

Now you can see in the above that the word

- **how** is likely to appear by 0.5 probability,
- **hello** is likely to appear by 0.3 probability,
- **is** is likely to appear by 0.2 probability.
  You are encouraged to use slight randomness when choosing the next word. If you don't, the predictions can fall into a loop of the same words.
  You could also use np.random.choice() here
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.choice.html

```
idx = np.random.choice(len(probabilities), p=probabilities)
return int_to_vocab[idx]
```

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

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

really good generated script 😊 How will it change on adding randomness in pick_word ?

DOWNLOAD PROJECT

RETURN TO PATH

Student FAQ