UDACITY

PROJECT

Generate TV Scripts

A part of the Deep Learning Nanodegree Foundation Program

PROJECT REVIEW

CODE REVIEW

NOTES

SHARE YOUR ACCOMPLISHMENT!

Meets Specifications

Great submission! 👍

You implemented the neural network correctly, trained the network to a low loss and made a generated script that is similar to the original TV script.

Required Files and Tests

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

All the unit tests in project have passed.

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)

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

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)

You have implemented all the placeholders with the right shapes and data types, well done!

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)

Good work!

We are handling the LTSM cells as a black box in our project, if you want to dive into the details I would recommend to read the following resource: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Note that it is not really necessary to have a dropout layer in here, as the goal of the project is to generate text and there is not performance metric so overfitting is not a real concern. Also when calculating the loss and generating the text you would have to set the keep probability to 1, as you want to use the full, not-stochastic network there.

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

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)

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)

Good work putting the previously defined functions together here!

To speed up the training of the network you could initialize the weights of the linear layer using (for example) the arguments

```
weights_initializer = tf.truncated_normal_initializer(stddev=0.1),
biases_initializer = tf.zeros_initializer()
```

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]

Your functions is correct, well done!

You could write it slightly more compact without a for loop using some Numpy trickery:

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

    x = np.array(int_text[: n_batches * batch_size * seq_length])
    y = np.array(int_text[1: n_batches * batch_size * seq_length + 1])
    y[-1] = x[0]

    xx = np.split(x.reshape(batch_size, -1), n_batches, 1)
    yy = np.split(y.reshape(batch_size, -1), n_batches, 1)

    return np.array(list(zip(xx, yy)))
```

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.**

Good choices for the hyperparameters:

1. The number of epochs is chosen such that the loss is sufficiently low.
2. Batch size and learning rate are chosen such that the network trains quickly.
3. The sequence length (16) is reasonable for the length of the sentences we want to generate - in the start of the exercise you calculated the average line length to be 11.5.
4. The embedding size and RNN size are big enough to handle the complexity of the sentences.

The project gets a loss less than 1.0

Your network's loss is well below 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

Note that you are able to easily recover the tensors after saving and loading the model, because you previously defined their names.

The `pick_word` function predicts the next word correctly.

Good job here!

I would suggest to add a bit of randomness using for example `np.random.choice` (https://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.random.choice.html) for two reasons:

1. Without randomness it is possible that the generation get's stuck in a loop, repeating the same words
2. It generates a unique script every time

Example implementation:

```python
def pick_word(probabilities, int_to_vocab):
    idx = np.random.choice(len(int_to_vocab),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.

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review

Student FAQ Reviewer Agreement