UDACITY

‹ Return to "Deep Learning" in the classroom

Generate TV Scripts

审阅 | 代码审阅 | HISTORY

Meets Specifications

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

Advanced tips for improving net results

- Try and use deeper architectures, which have general tendency to blow up or vanish the gradients - so there's a net architecture known as Residual Nets, used to circumnavigate the issues with deeper architectures
- Try using more fully connected layers or Bi-Directional LSTMs or GRUs to make the predictions even better
- Try and use more sophisticated methods like `lemmatisation` and `stemming` to create a more pruned vocabulary. Have a look at the NLTK library to understand more operations

If you are keen on learning a bit more into what Natural Language Scientists use regularly in their nets. Try reading up a bit more on

- Word2Vec Algorithm
- Glove Algorithm
- Sequence2Sequence tutorial

Keep up the good work !

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)

Good job there in implementing the function in such a sleek and concise way !

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

Perfect implementation of the `token_lookup` function ! Good job en making this correct

form previous submissions. Good job !

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)

Good job in implementing the placeholders correctly. Also, the fact that you used naming of tensors, where it was not mandated, that's a good practice !

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)

This was one of the relatively tougher parts of the entire project. Glad to see that you've implemented it so flawlessly. Very impressive indeed !

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)

All the layers have been implemented perfectly along with the careful implementation of linear activation function. Quite impressive !

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]

The 'get_batches` function seems to be working perfectly well in this case. Good implementation , keeping in mind the sequential ordering of inputs and outputs !

P.S. You can use the implementations suggested in previous review for more concise and easier implementation

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.

Great job on tuning hyperparameters nicely !

Number of epochs seems to be fine as the training loss has decreased steadily and seems to have stabilised. **Please note, a lower number of epochs would have been helpful, as it is clearly evident from the generated script that you're overfitting the training data**

Learning rate seems to be fine as well, though you can observe the loss fluctuating a bit - the trick is to decrease the lr when this happens. Generally , use a dynamic lr which say reduces by half if the training loss isn't improving.

Pro Tip : Given you've tuned your hyper parameters so well, you may want to read more about rational ways to tune all your hyper parameters by methods like random search, grid search etc. - http://neupy.com/2016/12/17/hyperparameter_optimization_for_neural_networks.html

The project gets a loss less than 1.0

The training loss is within the bounds - that is it's lower than 1. This is fantastic effort on your side.

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.

Good job on using randomness for choosing the next word. Your algorithm seems to be quite an unique one indeed. Brilliant that you actually took efforts and implemented this algorithm so nicely !

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

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

This looks like a TV Script, albeit an overfitted one :)

下载项目

返回 PATH