CS551H Natural Language Generation

Practical 7 – Word-level Neural NLG

In this practical, you will learn to build a few different word-level neural NLG models using the Keras deep learning library. The code samples in this practical have been tested on Google Colaboratory (Colab). You are free to use your personal computers if you have Keras installed on them.

Recall that the generation capability of a neural NLG model comes from a neural language model trained to predict the next token. In this practical we consider word-level tokens.

There are multiple ways the language modelling task can be framed at word-level. At the simplest level, the model could be trained on one-word-in and one-word-out sequences. We could then keep increasing the number of words in the input sequence to train on n-words-in and one-word-out sequences. We could also create sequences of different lengths by splitting the text on new line character and then tokenizing each line into word sequences that build up in length. This requires padding shorter sequences to bring all the input sequences to the same length. There is no single right way of framing the word-level language modelling problem. You need to match your NLG requirements with the capabilities of your language model.

Part 1. N-words-in and one-word-out

1. From MyAberdeen download word_level_lm.ipynb file and upload to Colab. This notebook has a simple program to build word-level LSTM models that work with n-word-in and one-word-out sequences. This module currently trains on a short rhyme and regenerates it. Run the program to generate text with different values of sequence_length which controls the value of n in n-words-in.
2. The program code is presented in a single cell. But it should not be hard to spot code corresponding to the critical steps of model building. Spend some time understanding the code corresponding to each of these steps. As with character-level model development, the first critical step is to make the language sequences into numerical tensors ready to be fed to LSTMs. Keras provides the Tokenizer class to encode language sequences into integer sequences. Learn how to work with the Tokenizer class.
3. You should then focus on the function define_model. You will find an Embedding layer added before adding an LSTM layer. This Embedding layer learns word embeddings as it trains. You could also use pretrained word embeddings such as GloVes, but in this practical you will train your own embeddings.
4. From MyAberdeen download republic_clean.txt file and upload to Colab using code from last practical. Make all the necessary changes to run the program to train on this larger corpus. Experiment with different values for sequence_length.
5. Experiment with tuning the model – change the number of layers and the number of units in these layers, running the model each time to see the impact of your changes on the output.
6. The Embedding layer currently set to create 10-dimensional word embeddings (output_dim set to 10). Experiment with other values of output_dim to learn about their impact on output quality.

Part 2. Working with padded sequences

7. From MyAberdeen download padded_word_lm.ipynb. This program is very similar to the one from part 1, except it works with padded sequences created from lines of source text. Keras provides support for padding sequences which you should learn to use. Run the program to generate text and compare its performance with the version from part 1.

Part 3. Further Exercises

8. Hand craft or select interesting examples of seed text to learn how seed text impacts output quality.
9. Experiment with simplified vocabulary – for example after removing some or all stop words.
10. Experiment with deeper models – adding more LSTM layers. Does your multi-layered model perform any better? This exercise is interesting because by defining a larger model you are defining a larger search space and therefore the model could learn lot more possibilities. But please bear in mind that our experiments are always limited by the size of our training data and the limitations of our hardware.

Part 4. Comparing character-level and word-level models

11. Using the code from this practical and the previous practical, create a program that builds both character-level and word-level models. Train and test these models on the same data and print output text on consecutive lines for ease of comparison. You may have to adjust the amount of training given to each model (particularly character-level model) for a fair comparison.
12. While there are applications (such as texting on smart phones) where good predictions of next word (or sequences of words) is important, think of other applications demanding many other NLG features such as semantic soundness, textual reports running into several pages etc. How suitable is the neural NLG technology as a whole (character-level or word-level) for such applications?