# Topics in Artificial Intelligence (CPSC 532S): Assignment #3: *RNNs for Language Modeling*

Due on Monday, February 4, 2017 at 11:59pm PST

In this assignment you will build a recurrent neural network (RNN)-based *encoder* and *decoder*. The decoder is also sometimes called a *neural language model*. Effectively you will build a sequence-to-sequence translation model that is very common in language translation (*e.g.*, translating from English to French). However, for simplicity and future use in Assignment 4, we will build a model that translates from English to English, in other words a language auto-encoder.

**Programming environment**

You will be using **PyTorch** for this assignment. If you are using Microsoft Azure credits provided, you will need to provision a Data Science Virtual Machine (VM) to use for the assignment. To do so, please follow the instructions here (clickable link):

Provisioning of the Data Science Virtual Machine for Linux on Microsoft Azure

If you provisioned the VM for the previous assignment, **DO NOT DO THIS AGAIN**; re-use the VM you created. PyTorch should be pre-installed and available in the Azure Data Science VM. **Make sure you are keeping track of your Azure credits.** Remember that you are being charged as long as VM is running, even if it is idle and not executing any jobs. It maybe best to do some of the coding and debugging locally (in CPU mode) and run on the VM only when you verified that your code works and you need GPU support. If you opt to do this assignment on your personal computer, it is your responsibility to install PyTorch and ensure it is running properly. A number of tutorials for doing this are available on-line.

**Data**

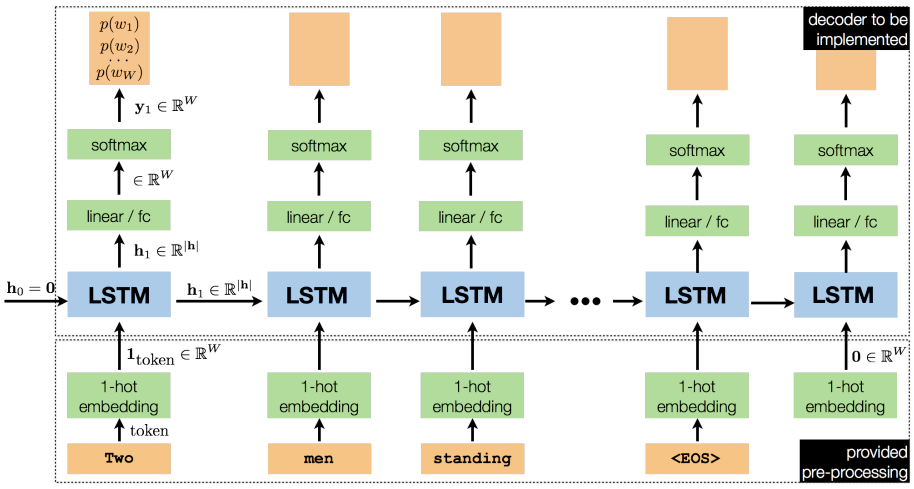
To train the models you will be using sentences that correspond to (approximately 20K) training images we used for Assignment 2 from MS-COCO dataset (20,000 sentences in total) and 500 sentences for validation (corresponding to 100 validation images). Note that the full MS-COCO contains approximately 400,000 sentences, with 5 sentences for each image. Here we sub-sampled the data for faster processing. You will be reusing the data from previous assignment.

## Assignment Instructions

You will need to follow instructions in the corresponding Jupyter notebook and submit the assignment as saved Jupiter notebook when done (including the results of executing code). As part of the assignment we are providing pre-processing code that tokenizes the text, defines a vocabulary of `vocabularySize` $= W = 1,000$ of most frequent words and represents each word as either 1-hot encoding ($\mathbf{x} \in \mathbb{R}^{1,000}$) or word2vec encoding ($\mathbf{x} \in \mathbb{R}^{300}$), where the length of the encoding vector we denote `wordEncodingSize`.

Once data is pre-processed you will follow the notebook through the following steps (as you do, you will find the Seq2seq PyTorch Tutorial extremely helpful; please ignore the parts discussing attention).

1. **Building Language Decoder.** First you will build a language model using RNN with LSTM units and 1-hot encoding for the words. The model is illustrated in the Figure below.



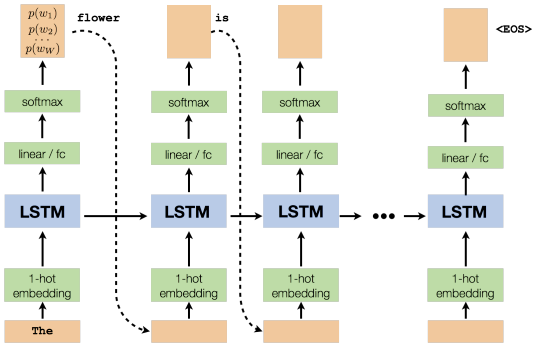

   The process and the sample code for building such a model is described in the PyTorch Seq2seq tutorial referenced above under "*Simple Decoder*", but you will need to use LSTM units instead of GRU and ignore starting from start-of-string `<SOS>` token. Starting with GRU implementation, as in the tutorial, may not be a bad starting point. The RNN you will build will need to take as input a tensor of sentences size (1 × `maxSequenceLength` × `wordEncodingSize`) and output the distribution over words (1 × `maxSequenceLength` × `vocabularySize`). In effect, at every time step of the RNN, the input is a current word and the output is the distribution over the following word. For the hidden state, use dimensionality of $|\mathbf{h}| = 300$.

2. **Training Language Decoder.** Once the language model is defined you will need to write the code to train it using the sentences. The process is described in the PyTorch seq2seq tutorial under "*Training the Model*". Start by using teaching forcing with `teacher_forcing_ratio = 1`. You will need to run the training with GPU support enabled as otherwise the training will be very slow. Make sure to use **Adam** optimizer and **CrossEntropyLoss**. Set the learning rate low enough so that you get convergence (you will want to monitor the loss every couple hundred samples or so, even though I am not explicitly asking for this).

3. **Building Language Decoder MAP Inference.** Inference in the model follows the training and is referred to as "*Evaluation*" in the PyTorch Seq2seq tutorial. The function should start from a single word, predict the distribution over the next word, pick the most likely next word from the distribution and pass it as the input to the next LSTM unit and so on until `<EOS>` tag is reached. The inference is illustrated in the Figure below:

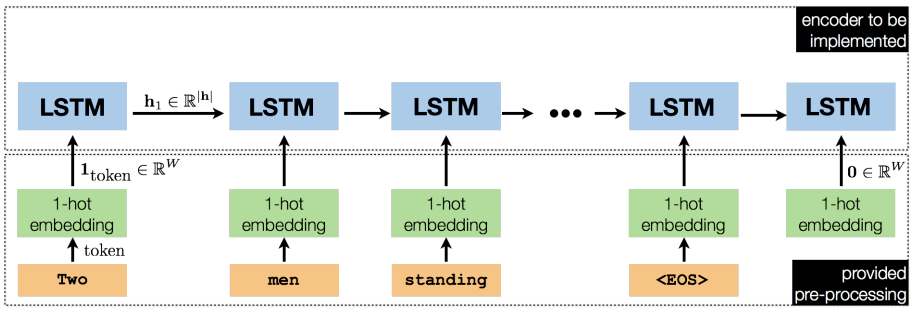
   To evaluate the trained model using the inference procedure, infer sentences that result by starting from `The`, `Man`, `Woman` and `Dog` (your notebook should show the 4 resulting sentences).

4. **Building Language Decoder Sampling Inference.** This is similar to above, but instead of picking the most likely word from the distribution predicted by the LSTM at every step, you should sample from the distribution according to the predicted probability. Evaluate the procedure by drawing 5 sample sentences that result by starting from `The`, `Man`, `Woman` and `Dog` (notebook should show 20 resulting sentences).




5. **Experiment with Teacher Forcing.** Now redo steps 2-4 with `teacher_forcing_ratio = 0.9` and `teacher_forcing_ratio = 0.8`. Comment on the results, including speed of convergence and the overall quality of results.

6. **Building Language Encoder.** Here you will follow the details in the Encoder portion of the PyTorch Seq2seq tutorial; again here you will need to replace GRU units by LSTM ones. The encoder should take the previous hidden state and one word at a time and progressively encode the sentence. The illustration of the model is shown in the Figure below. Again for the hidden state use dimensionality of $|\mathbf{h}| = 300$.




7. **Connecting Encoder to Decoder and Training End-to-End.** You will now connect the encoder to the decoder and train the Seq2seq encoder-decoder architecture end-to-end. Note that now, you will need to start the decoder from the `<SOS>` token and pass the last hidden state out of the encoder as the first hidden state of the decoder.

8. **Testing.** Take the validation set provided in the assignment that consists of 500 sentences from MS-COCO not used for training, for each run encoder then decoder using **MAP** inference. Report the average similarity between pairs of input and predicted output sentences using BLEU score (code for computing the score for pair of sentences is provided in the notebook).

9. **Encoding as Generic Feature Representation.** Use the final hidden state of the encoder to measure similarity between the first 10 validation sentences and the entire training set of 500,000K sentences. You would most likely want to pre-process the training set and save hidden states, so you do not need to re-compute those for finding nearest neighbor similarity.

10. **Effectiveness of word2vec.** Redo all steps (1)–(8) using the word2vec representation instead of 1-hot embedding.

**Optional [10 points]:** The entire assignment was done by training with one sample at the time. In practice, this is a bad idea and is typically much more difficult to get to converge. Usually you would

want to do this with mini-batches. This, however, is not trivial. For extra credit modify your code to support mini-batches. In essence, the pre-processing code will need to change to pad all sentences to a fixed maximum length `maxSequenceLength` of words (by padding sequences with zero vectors $\mathbf{0} \in \mathbb{R}^{1,000}$ or $\mathbf{0} \in \mathbb{R}^{300}$ respectively), which is convenient for training. Also the training would need to accept tensors of size `batch_size` $\times$ `maxSequenceLength` $\times$ `wordEncodingSize`) and output the distribution over words (`batch_size` $\times$ `maxSequenceLength` $\times$ `vocabularySize`).