# HW Assignment 5 (Due by 9:00am on April 12)

## 1 Theory (100 points)

Consider a binary input sequence $\mathbf{x} = x_1, x_2, ..., x_k, ...$, where $x_k \in \{0, 1\}$. Design a vanilla RNN whose output $y_k$ at position $k$ is 0 if the number of ones seen so far in $x_1, x_2, ..., x_k$ is even, and 1 if the number of ones seen so far is odd, for any position $k$. You are free to use *sigmoid*, *tanh*, or *ramp* as activation functions. Clearly show the RNN structure and the formulas used to compute the outputs and hidden states. Show the actual values of the parameters and prove that it produces the desired output. No points will be given if the complete proof is missing.

*Hint 1: You may have to use a multi-layer RNN.*

*Hint 2: The solution to a problem on a previous assignment can be very useful here.*

## 2 Textual Entailment using RNNs (200 points)

In this assignment, you will implement 2 LSTM models for textual entailment, with and without attention, based on the paper Reasoning about entailment with neural attention, by Rocktaschel et al., ICLR 2016. It is important that you read this paper before starting work on the assignment. Model 3, in particular, uses the same notation as in the paper.

Download the skeleton code from http://ace.cs.ohio.edu/~razvan/courses/dl6890/hw/hw05.zip. Follow the links on the course web site and 1) download the word embeddings that have been pre-trained using the word2vec package on the Google News corpus; 2) download the Stanford Natural Language Inference dataset. Put the two files in the `snli` folder.

Implement the 2 textual entailment models, `Model_2` and `Model_3`, as explained below, using PyTorch. Make sure that you organize your code in folders as shown in the table below.

```
dl6890/
   hw05/
      pytorch/
         train_rnn.py
         rnnExercise.py
         data_set.py
         README.txt
         output.txt
      snli/
         embedding.pkl
         snli_padding.pkl
```

Write code only in the file indicated in bold. The traces from running `python3 rnnExercise.py <mode>` for the 3 models should be saved in `output.txt`. In each step you will train the corresponding architecture and report the accuracy on the test data. By default the code uses the GPU if `torch.cuda.is_available()` returns true, otherwise it uses the CPU. It is recommended you run the code on an architecture with GPU, otherwise it can be very slow. It is also recommended that you read and understand the code in `rnnExercise.py` and `data_set.py`, besides the skeleton code in `train_.py`.

**Dataset**: Run `data_set.py`, which will create two files: 1) `embedding.pkl` containing embeddings only for words that appear in the SNLI dataset; 2) `snli_padding.pkl` containing the SNLI dataset wherein each sentence is a list of word IDs, padded with zeros up to the maximum sentence length.

The pickled embeddings in `embedding.pkl` contain 3 fields:

1. `word_to_id`: a Python dictionary that contains words to id mapping.

2. `id_to_word`: a Python dictionary that contains id to words mapping.

3. `vectors` : a NumPy 2D array than contains the embeddings for all the words in SNLI. The first row represents the embedding of the dummy word used for padding and is set to all zeros. The second row corresponds to the embedding of out of vocabulary (OOV) words. The rest of the vectors are for the words that exist in SNLI dataset. For each word, the row number is the word ID.

In order to process sentences in minibatches, all sentences are padded with 0 up to a maximum length and the true lengths of all the sentences are recorded for later use. Correspondingly, the SNLI dataset pickled in `snli_padding.pkl` contains 3 arrays – `train_set`, `dev_set`, and `test_set` – where each array is formed of 5 tensors: one tensor for the premise sentences, one tensor for the true lengths of the premise sentences, one tensor for the hypothesis sentences, one tensor for the true lengths of the hypothesis sentences, and one tensor for the labels of the sentence pairs (0 for entailment, 1 for neutral, 2 for contradiction).

**Model 1: Shared LSTM**: This model trains one LSTM model to process both premise and hypothesis. When processing a sentence, for each word in the sentence, the LSTM uses the word embedding as input and will multiply it internally with a parameter matrix to obtain a projected vector that has the same dimensionality as the LSTM state. The LSTM output at the end of the sentence is used as the representation of the sentence. Since sentences are padded with dummy words to have the same length, the code has to determine the LSTM output corresponding to the true length of the sentence. The premise and hypothesis representations will be concatenated and used as input to a feed-forward network with 3 hidden layers and one softmax layer. For regularization, the code uses dropout with the provided dropout rate.

**Model 2: Conditional LSTM (100 points)**: This model is described in section 2.2.1 in the ICLR paper. It uses two LSTM networks, one for the premise, one for the hypothesis. The cell state of the hypothesis LSTM is initialized with the value of the last cell state of the premise LSTM. The last output of the hypothesis LSTM is used as the representation of the sentence pair and fed as input to a feed-forward network with 3 hidden layers and one softmax layer. For regularization, use dropout with the provided dropout rate.

**Model 3: Conditional LSTM with Attention (100 points)**: This LSTM model is described in section 2.3 in the ICLR paper. It is like Model 2 above, but it also uses an attention mechanism, where the attention weights are computed between the last output of the hypothesis LSTM and all the outputs of the premise LTSM. Since sentences are padded with dummy words, a mask vector is computed for each premise sentence to indicate which words truly belong to the sentence and thus can be used when computing attention weights. The context vector computed with attention weights together with the last output of the hypothesis LSTM are linearly combined and passed through a *tanh* nonlinearity, as explained in the paper. The result is then used as the representation of the sentence pair and fed as input to a feed-forward network with 3 hidden layers and one softmax layer. For regularization, use dropout with the provided dropout rate.

For your convenience, complete code for Model 1 is provided in `train_.py`. You can reuse this code for the other two models, or you can create your own implementation. For both Model 2 and Model 3, you only have to write the forward propagation code.

## 2.1 Relevant PyTorch functionality

- `torch.nn.Module`: All 3 models need to be derived from this class.
- `torch.nn.Parameter`: When you define model parameters manually, as is thecase for the attention model parameters in Model 3, they need to be defined as `torch.nn.Parameter`, otherwise the corresponding variables will not be registered into the module, and hence they will not be trained.
- `torch.nn.LSTM`, `torch.nn.LSTMCell`: This are the PyTorch classes implementing LSTM's and LSTMCell's. You will need to use LSTMCell to implement the premise LSTM in Models 2 and 3, in order to be able to access the LSTM output for the actual last token in the sentence when the sentence is padded with dummy words. Examples of using LSTMCell are given in the documentation and this Time Series Prediction example.
- `torch.nn.functional.dropout()`, `torch.nn.Linear`: To apply dropout and implement linear layers.
- `torch.repeat` , `torch.expand`: Try to use these functions when you need to replicate a tensor over some dimension. The difference between torch.repeat and torch.expand is that torch.expand does not allocate memory for the extended elements.

The PyTorch tutorial on Machine Translation with a Sequence to Sequence Network and Attention is very relevant for this assignment, so it is recommended that you read it before you start writing code. Read also the documentation for the torch.nn.LSTM module.

For bonus points, you can try other configurations in order to get better results, such as a different number of fully connected layers between the LSTM and the softmax layer, different locations for inserting dropout layers, different activation functions for the fully connected layers, and other initialization schemes for the the manually created variables. You may also want to experiment with padding at the beginning instead at the end, and using the output from the last dummy token in the premise instead of the output from the actual last word in the premise.

## 3 Submission

Turn in a hard copy of your homework report at the beginning of class on the due date. Electronically submit on Blackboard a hw05.zip file that contains the hw05 folder in which you change code **only in the required files**. The screen output produced when running the rnnExercise.py code should be redirected to (saved into) the **output.txt** file. **Do not attempt to submit the snli data folder** on Blackboard!

On a Linux system, creating the archive can be done using the command:

```
> zip -r hw05.zip hw05.
```

Please observe the following when handing in homework:

1. Structure, indent, and format your code well.

2. Use adequate comments, both block and in-line to document your code.

3. On the theory assignment, clear and complete explanations and proofs of your results are as important as getting the right answer.

4. Make sure your code runs correctly when used in the directory structure shown above.