

# CMSC470 Project 3 Description

## Neural Machine Translation

### Part 1: Understanding the provided MT model

Prerequisite: Test out the installation by training the provided MT model on a small training set by running:

```
python3 main.py
```

This trains an RNN encoder-decoder model with attention and with default parameters defined in `main.py` for French to English translation. Using default settings, the training loss is printed every 1000 iterations so you can track progress. When training is done, the script prints the translations generated by the model for a random sample of the training data, as a sanity check.

There are 3 output files:

- `<output-dir>/args.pkl` -> Stores the model parameters
- `<output-dir>/encoder.pt` -> Stores the trained encoder model
- `<output-dir>/decoder.pt` -> Stores the trained decoder model

The same command if run with "`--eval`" flag (i.e. `python3 main.py --eval`) loads the trained model and runs it on the test file. The script translates examples from the provided test set, and evaluates the performance of the model by computing the BLEU score. The script also stores the output and reference translations to the file `<output-dir>/output.pkl`

On a CPU, running this to completion takes about 5-6 mins on our machines.

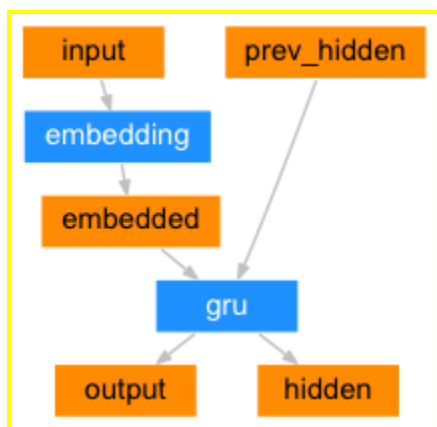
### Q1 Understanding the Training Data [12 pts]

Two training sets are provided: *eng-fra.train.small.txt* and *eng-fra.train.large.txt*. Data files are read in and processed by the `prepareData` function in `data.py`.

- Compute and report the following statistics for each of the training set (after processing by prepareData):
  - the number of training samples
  - the number of English word types and tokens
  - the number of French word types and tokens
  - the average English sentence length
  - the average French sentence length
- Based on these statistics only (no knowledge of French needed), describe one way in which English and French are different, and how it might impact machine translation performance.

## Q2 Understanding the Model: Encoder [8 pts]

A RNN encoder is implemented in class Encoder in model.py. The Encoder class defines a RNN computation graph using [Tensors](#), which define nodes in the graph and will hold the values of learnable parameters in the network, and Functions which define the network structure by defining how output Tensors are computed as functions of input Tensors (i.e. forward propagation). Here the encoder uses a [Gated Recurrent Unit \(GRU\)](#), which is a variant of the simple RNN designed to address the vanishing gradient problem. For every input word, the encoder produces an output vector and a hidden state, and uses the hidden state as input to encode the next input word, as illustrated below:



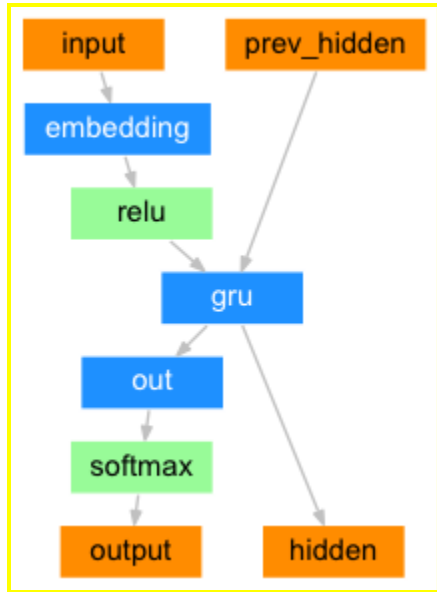
Given default parameter settings

- What are the dimensions of the word embedding matrix for the encoder?
- How many trainable parameters does the GRU have?

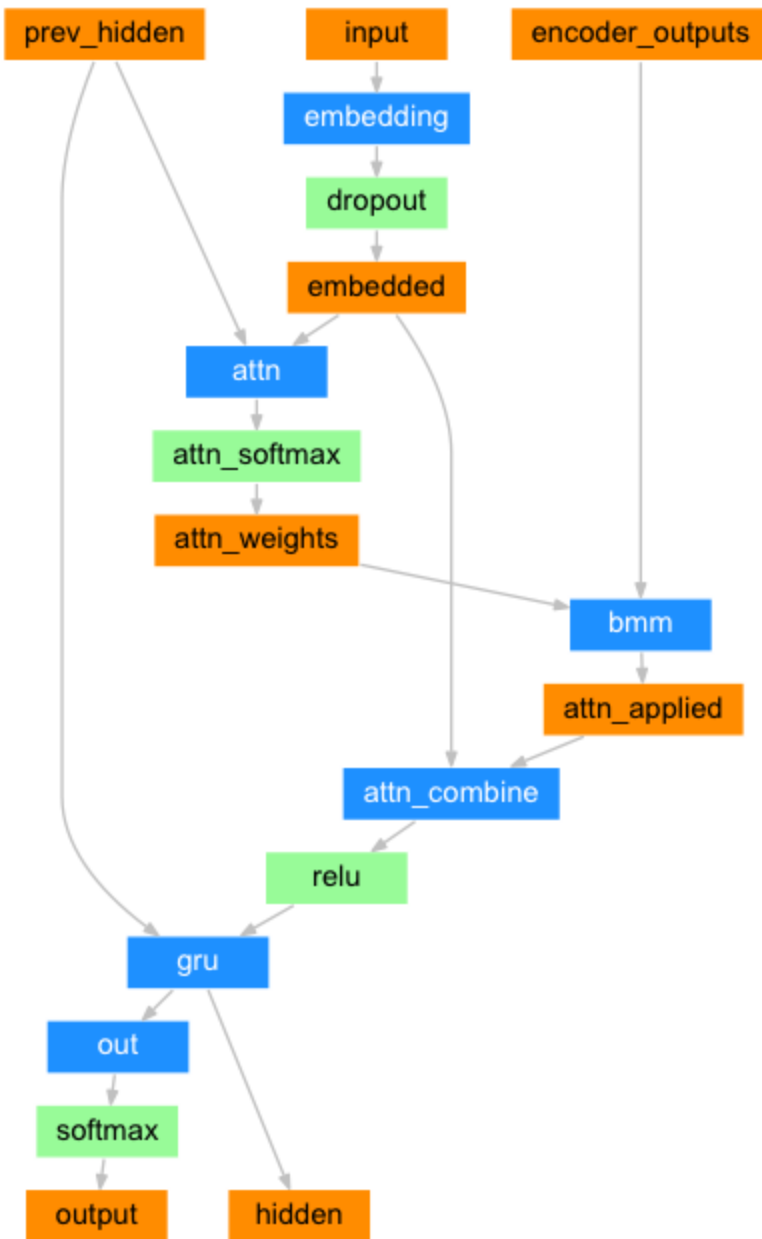
### Q3 Understanding the Model: Decoders [11 pts]

Two decoders are provided.

SimpleDecoder implements a left-to-right RNN decoder based on GRU, as illustrated below:



The class AttnDecoderRNN implements a more sophisticated decoder, which uses an attention network to compute an attention-weighted representation of the input sentence for each decoding time-step as illustrated below:



- Based on the provided code and consulting the pytorch documentation as needed, provide the complete mathematical formula used to compute the weighted sentence representation vector (in `attn_applied`) as a function of encoder and decoder hidden state vectors, stored in `encoder_outputs`, `previous_hidden`, and `embedded` (i.e., no need to include the dropout layer in your function).

- Train and evaluate a model that does not use attention. This can be done by setting the "simple" flag in main.py. You need to provide code in code.py for define\_simple\_decoder and run\_simple\_decoder. Report its BLEU score on the test set. **The code itself will be graded in Part 2 of the project.**

## Q4 Understanding Training [10 pts]

The EncoderDecoder class (in seq2seq.py) encapsulates the training and decoding procedures. Given default parameters, provide the following information about the training process:

- Give the mathematical formula for the loss function when computed on the following set of training examples:  $\{(F_1, E_1), \dots (F_n, E_n)\}$
- Does the loss function incorporate a bias against short outputs?
- Given a computation graph, most of the training work is done under the hood by [autograd](#), pytorch's auto-differentiation package. Training is controlled by 3 steps: (1) compute the gradient of the loss w.r.t. model parameters and store gradient values, (2) update parameters and (3) zero out stored gradient values before computing the gradient again on new examples. Report the line number(s) corresponding to the implementation of each step in the train function (make sure to get line numbers in the original unmodified file.)
- Does the training algorithm update parameters after each sample (online), after seeing the entire training set (batch), or after seeing a mini-batch of samples?

## Q5 Understanding Decoding [8 pts]

The evaluate function produces translations for new inputs using greedy search.

Consider the following French sentences:

1. "L'anglais n'est pas facile pour nous." (English is not easy for us)
2. "J'ai dit que l'anglais est facile." (I said that English is easy)
3. "Je fais un blocage sur l'anglais." (I get stuck with English)
4. "Je n'ai pas dit que l'anglais est une langue facile." (I didn't say that English is an easy language)

Translate these sentences using the encoder-decoder with attention, trained on the large training set with default parameters (*hint: after training the model, just uncomment lines*

111-114 in `main.py` and run on eval mode). For each sentence, describe and explain (briefly) the behavior of the decoder, specifically: is the decoder able to translate this sentence? Or if it terminates with an error? Give the translation and explain why it is good or bad. If no, explain why not. To help you interpret the model output, you can visualize the attention weights using `evaluateAndShowAttention` which is defined in the class `EncoderDecoder` in `main.py`.

## Q6 Training Configuration [6 pts]

For the LR (learning rate) hyperparameter:

- Briefly explain its role
- Try setting `--lr` to **0.09** and **0.03** and describe the impact on (a) training loss and (b) test BLEU

# Part 2: Improving the Model Architecture

## 2.0 Implementing the simple encoder [5 pts]

Provide the implementation of `define_simple_decoder` and `run_simple_decoder` that you wrote to answer question Q3 in Part 1.

## 2.1 Bidirectional encoder [10 pts]

Implement a bidirectional encoder in the class `BidirectionalEncoderRNN` (in `code.py`). Train a model using it (run `python3 main.py --bidirectional`) and evaluate its impact on translation quality by computing its BLEU score.

- Provide your implementation in `code.py`  
(For your reference, running our implementation on our machine gives a BLEU score of about 0.14.)

## 2.2 Attention Models [30 pts]

Implement a new decoder in class `AttnDecoderRNNDot` (in `code.py`) that uses dot product-based attention. (run `python3 main.py --dot` to train your model) [15 pts]

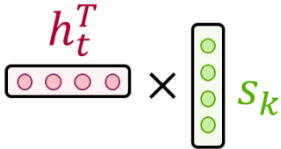
- Provide your implementation of class `AttnDecoderRNNDot` in `code.py`  
(For your reference, running our implementation on our machine gives a BLEU score of about 0.13)

Implement a new decoder in class `AttnDecoderRNNBilinear` (in `code.py`) that uses multiplicative attention. (run `python3 main.py --bilinear` to train your model) [15 pts]

- Provide your implementation of class `AttnDecoderRNNBilinear` in `code.py`  
(For your reference, running our implementation on our machine gives a BLEU score of about 0.12)

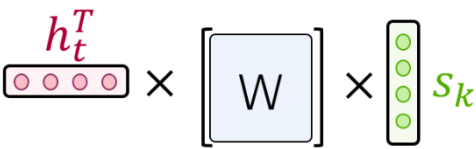
**Hint:**

Dot-product



$$\text{score}(h_t, s_k) = h_t^T s_k$$

Bilinear



$$\text{score}(h_t, s_k) = h_t^T W s_k$$

Refer to the default attention class implement in `model.py` and the figure above when implementing the attention classes.