Computer Science 401/2511 5 Feb. 2026
St. George Campus University of Toronto

Homework Assignment #2
Due: March 5, 2026,

Neural Machine Translation (MT) using Transformers

Email: `csc401-2026-01-a2@cs.toronto.edu`. Please prefix your subject with `[CSC401_W26-A2]`.
TA:Tian Yu
Instructor: Gerald Penn, Ken Shi

Building the Transformer from Scratch

The transformer architecture is the building block that has fueled nearly every headline innovation in NLP for the last six years. But despite the very large number of people who work with neural NLP, only relatively few understand its internal workings. In this assignment, you will build a transformer model from beginning to end, and train it to do some basic but effective machine translation tasks using the Canadian Hansards data. In §1, we will guide you through the process of implementing all the components of a transformer model. In §2, you will put together a transformer with those components. §3 discusses greedy and beam search decoders for target sentence generation. In §4, you will train and evaluate the model. Finally, in §5, you will use the trained mode to do some real machine translation and write up a report that analyzes your output.

Goal By the end of this assignment, you will have acquired a low-level understanding of the transformer architecture and implementation techniques of the entire data processing → training → evaluation pipeline of an MT application.

Starter Code The starter code of this assignment is distributed through MarkUs. The training data are located at `/u/cs401/A2/data/Hansard` on `teach.cs`.

We use Python version 3.12.3 on `teach.cs`. That is, just run everything with the `python3.12` command. You may need to add `srun` commands to request computational resources — please follow the instructions in the following sections to proceed. You should not need to set up a new virtual environment or install any packages on `teach.cs`. You can work on the assignment on your local machine, but you must make sure that your code works on `teach.cs`. Any test cases that fail due to incompatibility issues will not receive partial marks.

1 Transformer Building Blocks and Components [12 Marks]

We begin with the three basic building blocks of a transformer: the layer norm, multi-head attention and feed-forward modules (a.k.a. MLP weights).

LayerNorm The normalization layer computes the following. Given an input representation $\mathbf{h}$, the normalization layer computes its mean μ and the standard deviation σ. Then, it outputs the normalized features.

$$\mathbf{h} \leftarrow \frac{\gamma(\mathbf{h} - \mu)}{\sigma + \varepsilon} + \beta \tag{1}$$

Using the instructions, please complete the `LayerNorm.forward` method.

FeedForwardLayer The feed-forward layer is a two-layer fully connected feed-forward network. As shown in the following equation, the input representation $\mathbf{h}$ is fed through two layers of fully connected layers. Dropout is applied after each layer, and ReLU is the activation function.

$$\begin{aligned} \mathbf{h} &\leftarrow \text{dropout}(\text{ReLU}(\mathbf{W}_1\mathbf{h} + \mathbf{b}_1)) \\ \mathbf{h} &\leftarrow \text{dropout}(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2) \end{aligned} \tag{2}$$

Using the instructions, please complete the `FeedForwardLayer.forward` method.

MultiHeadAttention Finally, you need to implement the most complicated but important component of the transformer architecture: the multi-head attention module. For the base case where there's only $H = 1$ head, the attention score is calculated using the regular cross-attention algorithm:

$$\text{dropout}(\text{softmax}(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}))\ \mathbf{V} \tag{3}$$

Using the instructions, please complete the `MultiHeadAttention.attention` method.

Then, you need to implement the part where the query, key and value are split into H heads, and then pass them through the regular cross-attention algorithm you have just implemented. Next, you should combine the results. Don't forget to apply the linear combination and dropout when you output the final attended representations.

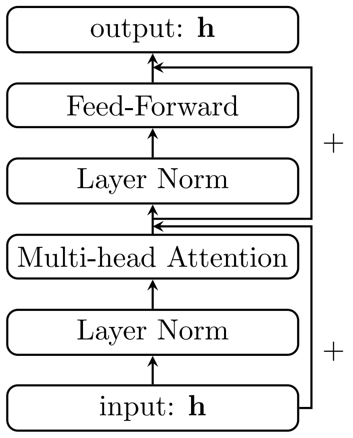
Using the instructions, please complete the `MultiHeadAttention.forward` method.

2 Putting the Architecture (Enc-Dec) Together [20 Marks]

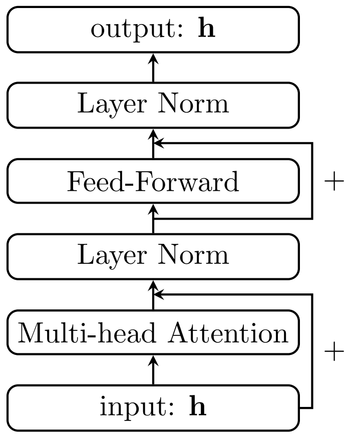
Once you have the building blocks, put everything together and create the full transformer model. We will start from a single transformer encoder layer and a single decoder layer. Next, we build the complete encoder and the complete decoder together by stacking the layers together. Finally, we connect the encoder with the decoder and complete the final transformer encoder–decoder model.

TransformerEncoderLayer You need to implement two types of encoder layers. Pre-layer (Figure 1a) and post-layer (Figure 1b), as their names suggest, apply layer normalization before/after the representation is fed into the following multi-head attention module.

Using the instructions, please complete the `pre_layer_norm_forward` and the `post_layer_norm_forward` methods of the `TransformerEncoderLayer` class.



(a) Pre-layer normalization for encoders.



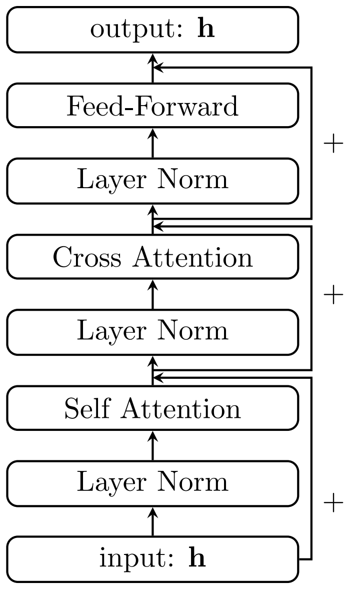
(b) Post-layer normalization for encoders.

Figure 1: Two types of `TransformerEncoderLayer`.

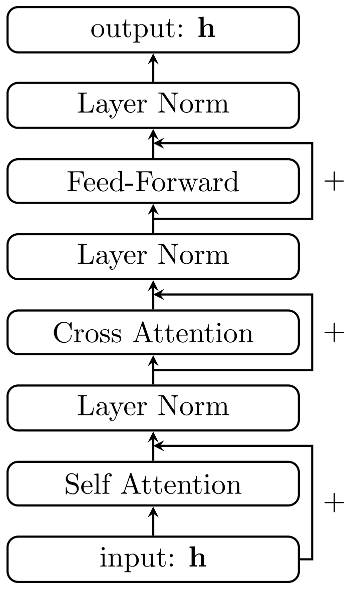
TransformerEncoder You don't need to implement the encoder class yourself; the starter code already contains the implementation. You read it, however, as it will a good reference for the following tasks.

TransformerDecoderLayer Again, implement both pre- and post-layer normalization. Recall from lecture that there are two multi-head attention blocks for decoders. The first one is a self-attention block and the second one is a cross-attention block.

Using the instructions, complete the `pre_layer_norm_forward` and the `post_layer_norm_forward` methods of the `TransformerDecoderLayer` class.



(a) Pre-layer normalization for decoders.



(b) Post-layer normalization for decoders.

Figure 2: Two types of `TransformerDecoderLayer`.

TransformerDecoder Similar to `TransformerEncoder`, you should pass the input through all the decoder layers. Make sure to add the LayerNorm module in the correct place depending if the model is pre- or post-layer normalization. Finally, don't forget to use the logit projection module on the final output.

Using the instructions, please complete the `TransformerDecoder.forward` method.

TransformerEncoderDecoder After ensuring that the encoder and the decoder have both been built properly, it is time to put them together. You need to implement the following methods:

- The method `create_pad_mask`, a helper function to pad a sequence to a specific length.
- The method `create_causal_mask`, a helper function to create a so-called "causal" (upper) triangular mask.
- After finishing the two helper methods, you can implement the `forward` method that connects everything. In particular, you first create all of the appropriate masks for the inputs. Then, you feed them through the encoder. And, finally, you obtain the final result by feeding everything through the decoder.

You will need to determine the tensor shapes for all functions yourself, given their inputs and outputs. The shapes of the tensors returned from the masking functions are a hint towards this, but you may need to reshape tensors (see `torch.reshape`).

3 MT with Transformers: Greedy and Beam-Search [20 Marks]

The `a2_transformer_model.py` file contains all of the functions that must be completed along with detailed instructions (with hints). Here, we list the high-level methods/functions that you need to complete.

3.1 Greedy Decode

You can warm up with the greedy algorithm. At each decoding step, compute the (log) probability over all the possible tokens. Then, choose the output with the highest probability and repeat the process until all the sequences in the current mini-batch terminate.

Using the instructions, complete the `TransformerEncoderDecoder.greedy_decode` method.

3.2 Beam Search

The hardest part of the assignment is probably beam search. But don't worry: we have broken everything down into smaller, and much simpler chunks. We will guide you step by step to complete the entire algorithm.

Beam search is initiated by a call to the `TransformerEncoderDecoder.beam_search_decode` method. Recall from lecture that its job is to generate *partial translations* (or, *hypotheses*) from the source tokens during the decoding phase. So, the `beam_search_decode` method i called whenever you try to decode candidate translations (e.g. from `TransformerRunner.[translate, compute_average_bleu_over_dataset]` etc.).

Complete the following functions in the `TransformerEncoderDecoder` class:

1. `initialize_beams_for_beam_search`: This function will initialize the beam search by taking the first decoder step and using the top-k outputs to initialize the beams.
2. `expand_encoder_for_beam_search`: Beam search will process 'batches' of size 'batch_size * k' so we need to expand the encoder outputs so that we can process the beams in parallel. (*Tip*: You should call this from within the preceding function).

3. `repeat_and_reshape_for_beam_search`: this is a relatively simple expand and reshaping function. See how it is called from the `.beam_search_decode` method and read the instructions in the function's comments. (*Tip*: Review Torch.Tensor.expand).

4. `score_sequence_for_beam_search`: This function will get the score of each sequence by summing the log probabilities. See how it is called from the `.beam_search_decode` method and read the instructions in the function's comments.

5. `finalize_beams_for_beam_search`: Finally, this functions as its name - it will take a list of top beams of length batch_size, where each element is a tensor of some length and return a padded tensor of the top beams. It returns the ultimate result of the `.beam_search_decode` method – see how the return value is consumed by the calling functions (e.g. 'translate').

4 MT with Transformers: Training and Testing [20 Marks]

4.1 Calculating BLEU scores

Modify `a2_bleu_score.py` to be able to calculate BLEU scores on single reference and candidate strings. We will be using the definition of BLEU scores from the lecture slides:

$$BLEU = BP_C \times (p_1 p_2 \dots p_n)^{(1/n)}$$

To do this, you will need to implement the functions `grouper(...)`, `n_gram_precision(...)`, `brevity_penalty(...)`, and `BLEU_score(...)`. Make sure to **carefully** follow the doc strings of each function. Do not re-implement functionality that is clearly performed by some other function.

Your functions will operate on sequences (e.g., lists) of tokens. These tokens could be the words themselves (strings) or integer IDs corresponding to the words. Your code should be agnostic to the type of token used, though you can assume that both the reference and candidate sequences will use tokens of the same type.

Please scale the BLEU score by 100.

4.2 The Training Loop

Before you start working on this part of the assignment, have a good look at the `train` function. It describes how the training loop works. For each epoch, we first train the model using all the data from the training set. Then, we evaluate the model's performance upon the completion of the epoch. We repeat the process until we reach the specified maximum epoch number. For this part of the assignment, you will need to implement the following methods:

`train_for_epoch` The training of one epoch contains seven steps:

1. We iterate through the training dataloader to obtain the current mini-batch.

2. Then, we send the data tensors to the appropriate devices (CPU or GPU).

3. Call `train_input_target_split` to prepare the data for teacher-forcing training.

4. Feed the data through the transformer model and collect the logits.

5. Compute the loss value using the loss function.

6. Call `loss.backward()` to compute the gradients.

7. Call `train_step_optimizer_and_scheduler` to update the model using the optimizer, and step the scheduler.
   Note: You should handle gradient accumulation (using the `accum_iter` parameter) for full marks.

`train_input_target_split` This method splits target tokens into input and target for maximum likelihood training (teacher forcing).

`train_step_optimizer_and_scheduler` This method steps the optimizer, zeros out the gradient, and steps the scheduler.

`compute_batch_total_bleu` This function computes the total BLEU score for each n-gram level over elements in a batch. **Note**: You need to clean up the sequences by removing **ALL** special tokens (`sos_idx`, `eos_idx`, `pad_idx`).

4.3 Run Model, Run!

OK, enough coding - time for some model training and deployment. In this part of the assignment, you will (1) train models with different settings, (2) evaluate those models on the hold-out test set, and (3) deploy the model and do some actual translation.

4.3.1 Training the Models

Debugging and Development We provide a `--tiny-preset` option to make your model train only on a tiny sliver of the dataset. You should use this option with a smaller, toy model for debugging during your development phase. Training this toy model using the CPU on `teach.cs` will take approximately 10 seconds to complete. **Note: this is for developing the initial code only and you cannot expect good performance by training the tiny model.** We advise using the real dataset to debug problems related to training. The following command is just an example. You should modify the command to test different components of your code.

```
srun -p csc401 --pty \
    python3 a2_main.py train \
        model_test.pt \
        --tiny-preset \
        --source-lang f \
        --with-transformer \
        --encoder-num-hidden-layers 2 \
        --word-embedding-size 20 \
        --transformer-ff-size 31 \
        --with-post-layer-norm \
        --batch-size 5 \
        --gradient-accumulation 2 \
        --skip-eval 0 \
        --epochs 2 \
        --device cpu
```

The `srun -p csc401 --pty` part of the command requests a compute node on teach.cs. This will provide you with dedicated CPUs and reduce congestion during peak times. The `--pty` flag enables pseudo-terminal mode, which allows your breakpoints to work properly. But if you want to run the assignment locally, you should remove this.

Debugging Tips and Hints We have provided the functions `nan_check` and `inf_check` in `a2_utils.py` to help you catch NAN and INF errors. NANs are common issues that you may encounter during training. These are often the result of accidentally calculating an INF value (often as a final loss or incorrectly masked attention value). This then results in a parameter update with an INF value, which in turn causes the parameters to become NAN *in the next training step.* These will then propagate any time those parameters are used. Thus if you encounter NAN values, you should check for INF values in the previous step.

There are two separate stages: training and evaluation. These can be hard to debug as they depend on each other; the training stage does intermediate evaluations and the evaluation stage requires a trained model. To help you determine where potential bugs are, first consider the log-loss. If the loss is very high or not decreasing, more often than not, the bug will be in a) the attention mechanism or b) the training loop. If the loss looks good but the BLEU score does not, the issue is likely to be in the inference functions.

We will not tell you acceptable ranges of the log-loss and BLEU score in terms of grading, however - you need to determine those for yourself. Ask yourself: What would a very low or high BLEU score mean? What would the log-loss be for a random guess (given the vocabulary size)? But we will provide you some output from our own development to guide you. These values will obviously not be the same values that your model produces.

```
[Device:cuda] Epoch 1 Training ====
Forward Step:      1/  1086 | Accumulation Step:   0 | Loss:   9.88 | Learning Rate: 8.5e-06
Forward Step:    201/  1086 | Accumulation Step:  20 | Loss:   9.21 | Learning Rate: 1.8e-04
Forward Step:   1001/  1086 | Accumulation Step: 100 | Loss:   6.50 | Learning Rate: 8.6e-04
[Device:cuda] Epoch 1 Validation ====
Epoch 1: loss=6.307405866312059, BLEU: skipped until epoch 4, time=00:02:05
[Device:cuda] Epoch 2 Training ====
Forward Step:      1/  1086 | Accumulation Step:   0 | Loss:   3.87 | Learning Rate: 9.4e-04
Forward Step:    201/  1086 | Accumulation Step:  20 | Loss:   3.56 | Learning Rate: 1.1e-03
Forward Step:   1001/  1086 | Accumulation Step: 100 | Loss:   2.85 | Learning Rate: 1.8e-03
[Device:cuda] Epoch 2 Validation ====
Epoch 2: loss=2.798794230923029, BLEU: skipped until epoch 4, time=00:04:12
[Device:cuda] Epoch 3 Training ====
Forward Step:      1/  1086 | Accumulation Step:   0 | Loss:   1.99 | Learning Rate: 1.9e-03
Forward Step:    201/  1086 | Accumulation Step:  20 | Loss:   2.03 | Learning Rate: 2.0e-03
Forward Step:   1001/  1086 | Accumulation Step: 100 | Loss:   1.95 | Learning Rate: 2.5e-03
[Device:cuda] Epoch 3 Validation ====
Epoch 3: loss=1.9403861648030922, BLEU: skipped until epoch 4, time=00:06:22
[Device:cuda] Epoch 4 Training ====
Forward Step:      1/  1086 | Accumulation Step:   0 | Loss:   1.53 | Learning Rate: 2.4e-03
Forward Step:    201/  1086 | Accumulation Step:  20 | Loss:   1.58 | Learning Rate: 2.4e-03
Forward Step:   1001/  1086 | Accumulation Step: 100 | Loss:   1.58 | Learning Rate: 2.1e-03
[Device:cuda] Epoch 4 Validation ====
Epoch 4: loss=1.5846078365108145, BLEU-4: 35.4983 BLEU-3: 42.2861, time=00:09:06
[Device:cuda] Epoch 5 Training ====
Forward Step:      1/  1086 | Accumulation Step:   0 | Loss:   1.37 | Learning Rate: 2.1e-03
Forward Step:    201/  1086 | Accumulation Step:  20 | Loss:   1.30 | Learning Rate: 2.1e-03
Forward Step:   1001/  1086 | Accumulation Step: 100 | Loss:   1.34 | Learning Rate: 1.9e-03
[Device:cuda] Epoch 5 Validation ====
Epoch 5: loss=1.3434471010503188, BLEU-4: 36.0997 BLEU-3: 42.8254, time=00:11:48
Finished 5 epochs
```

Gradient accumulation is correctly implemented when the model produces the same results after having halved the batch size and doubling the amount of accumulated steps. We advise that this be implemented after the model and basic training loop have been correctly implemented.

Training the Final Models with GPUs After ensuring that your code works properly, you should train your model using both the pre- and post-layer normalization settings. Each epoch of training should take approximately 2 minutes. This time may vary depending on GPU availability, however.

Note: You are not required to train the model with post-layer normalization, but you must ensure that post-layer normalization is implemented correctly. The correctness of both pre- and post-layer normalization will have equal importance during the evaluation of the code. For the analysis and training, only the pre-layer normalization setting is required.

```
# Train the model using pre-layer-norm
srun -p csc401 --gres gpu \
    python3 -u a2_main.py train \
    transformer_model.pt \
    --device cuda
```

Then, include a printout of the the training logs in `analysis.pdf`. You have the option to use WandB for this part of the assignment. See Appendix B for more information.

4.3.2 Evaluate the Models

Now, evaluate the models using the following commands:

```
# Test the model using pre-layer-norm
srun -p csc401 --gres gpu \
    python3 -u a2_main.py test \
    transformer_model.pt \
    --device cuda
```

5 Analysis [8 Marks]

You are done! Now that you have completed all the parts needed for **training** your transformer end-to-end, as well as **testing** it, you're ready to finish this assignment with some **analysis**. Report the evaluation results from the preceding section in your latex report file: `analysis.pdf`.

Is there a difference between BLEU-3 and BLEU-4? What do you think could be the reason behind the differences? Write your answer in `analysis.pdf`.

5.1 Translate Some Sentences

In the `TransformerRunner.translate` method, you are tasked with processing a "raw" input sentence through several stages to obtain its translation. You need to (1) tokenize the sentence, (2) convert tokens into ordinal IDs, (3) feed the IDs into your model, and (4) finally, convert the output of the model into an actual sentence. You can load your trained model with the following command.[1]

```
srun -p csc401 --pty python3 a2_main.py interact transformer_model.pt
```

An interactive Python prompt will start and your can begin translating with the function `translate`. You can then utilize the model by interacting with it in the prompt as shown.

```
Trained model from path Your Model.pt loaded as the object 'model'
Python 3.12.3 (main, May  2 2024, 13:31:38) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
```

[1]It's OK to run the model with the greedy decoder if you haven't completed the implementation of beam search. You won't incur any penalty for doing so. To enable greedy decoding, simply add a `--greedy` flag at the end of the command.

```
>>> translate("Mon nom est Frank.")
'<s> frank is frank frank frank </s>'
```

For this part of the assignment, you will translate a few sentences from French into English using your model, using a fine-tuned, pre-trained transformer model (T5 MT model or Bart MT model), and using a large, established model (Google Translate or ChatGPT). First, you need to translate the eight sentences in `a2_sentences.txt` [2] into English using your model.

Then, translate the same sentences into English using your choice of the T5 MT model or the Bart MT model. Refer to the Colab notebook for usage. Next, translate the same eight sentences into English using your choice of Google Translate or ChatGPT.

In Section 2, "Translation Analysis," of `analysis.pdf`, list all the translations and answer the following questions.

1. Describe the **overall** quality of the three types of models. Which one is the best and which one is the worst?
2. What attributes and factors of the models do you think play a role in determining the quality?
3. Now, focus on the quality of your model's translations of individual sentences. Which sentences does your model translate better, and which does it translate worse? Can you identify a pattern? Describe the pattern of quality across different types of sentences.
4. What about the fine-tuned, pre-trained model and Google Translate/ChatGPT's quality for individual sentences? Does the previous pattern still persist? Why or why not?

Submission Requirements

This assignment is submitted electronically via MarkUs. You should submit a total of five ("5") required files as follows:

1. Your code: `a2_transformer_model.py`, `a2_transformer_runner.py`, and `a2_bleu_score.py`.
2. Your analysis: `analysis.pdf`. N.B. a *latex* template for this file has been provided to you in the starter code as `a2_report.zip`.
3. `ID.txt`: Provide pertinent information as per the `ID.txt` template on the course's website, including this statement:

   > "By submitting this file, I declare that my electronic submission is my own work, and accords with the University of Toronto Code of Behaviour on Academic Matters and the Code of Student Conduct, as well as the collaboration policies of this course."

You do not need to hand in any files other than those specified above.

[2] French accent marks (é,ç,à,ô,ï) are removed in the Canadian Hansards dataset and, therefore, you should too. Use *francaise*, *Universite* and *etudiants* instead of *française*, *Université* and *étudiants*.

Appendices

A Canadian Hansards

The main corpus for this assignment comes from the official records (*Hansards*) of the 36th Canadian Parliament, including debates from both the House of Representatives and the Senate. This corpus is available at `/u/cs401/A2/data/Hansard/` and has been split into `Training/` and `Testing/` directories.

This data set consists of pairs of corresponding files (`*.e`, the English equivalent of the French `*.f`) in which every line is a sentence. Here, sentence alignment has already been performed for you. That is, the n^{th} sentence in one file corresponds to the n^{th} sentence in its corresponding file. Note that these data only consist of sentence pairs; no alignments, many-to-one, many-to-many, or one-to-many, are included.

B Visualizing and logging training with WandB

You have the option to use '*Weights and Biases*' [W&B] to visualize and log your model training. Go to the W&B site[3] and sign-up, then create a new project space named: '`csc401-W26-a2`'. We refer to your W&B *username* as `$WB_USERNAME` hereafter. You need to log in to W&B using the command line by [4] :

1. Generate an API key: https://wandb.ai/authorize
2. In the terminal, set the API key as an environment variable by:

```
export WANDB_API_KEY="<Step 1 API Key>"
```

3. CLI login to W&B by:

```
wandb login
```

Then, add the `--viz-wandb $WB_USERNAME` flag to your model training commands.

C Suggestions

C.1 Check Piazza regularly

Updates to this assignment as well as additional assistance outside tutorials will be distributed primarily through Piazza (`https://piazza.com/class/m0hmhakihyc4sz`). **It is your responsibility to check Piazza regularly for updates.**

C.2 Run cluster code early and at irregular times

Because GPU resources are shared with your classmates, your `srun` job may end up on a weaker machine or even postponed until more resources are available, if too many students are training at once. To help balance resource usage over time, we recommend that you finish this assignment as early as possible. You might find that your peers are more likely to run code at certain times in the day. To check how many jobs are currently queued or running on our partition, please run `squeue -p csc401`.

[3]https://wandb.ai/

[4]https://github.com/wandb/server/issues/75#issuecomment-3749956143

If you decide to run your models right before the assignment deadline, please be aware that we will be unable to request more resources or run your code sooner. **We will not grant extensions for this reason.**