11-751/18-781 Speech Recognition and Understanding (Fall 2023) Coding Assignment 3

OUT: October 11th, 3:30 PM ET
DUE: November 3rd, 11:59 PM ET

Instructor: Prof. Shinji Watanabe
TAs: Xuankai Chang, Brian Yan, Yifan Peng

Collaboration Policy

Assignments must be completed individually. You are allowed to discuss the homework assignment with other students and collaborate by discussing the problems at a conceptual level. However, your answers to the questions and any code you submit must be entirely your own. If you do collaborate with other students (i.e. discussing how to attack one of the programming problems at a conceptual level), you must report these collaborations. Your grade for Coding Assignment 3 will be reduced if it is determined that any part of your submission is not your individual work.

Collaboration without full disclosure will be handled in compliance with CMU Policy on Cheating and Plagiarism: https://www.cmu.edu/policies/student-and-student-life/academic-integrity.html

Late Day Policy

You have a total of **5 late days (i.e., 120 hours)** that you can use over the semester for the assignments. If you do need a one-time extension due to special circumstances, please contact the instructor (Shinji Watanabe) via Piazza.

Programming Notes for Coding Assignment 3

Set up python environment

Suppose you use anaconda to manage python environments. We provided a script which sets up a new anaconda instance and environment. You may choose your own package management solution if you'd like.

```
# use the provided script to setup a new environment
cd 11751_hw3_handout
./setup_anaconda.sh miniconda ctc-asr 3.9
. ./activate_python.sh

# install pytorch and other packages
conda install pytorch==1.12.1 torchaudio==0.12.1 cudatoolkit=11.6 -c pytorch -c conda-forge
pip install -r requirements.txt
```

Code template

The code template has the following directory structure:

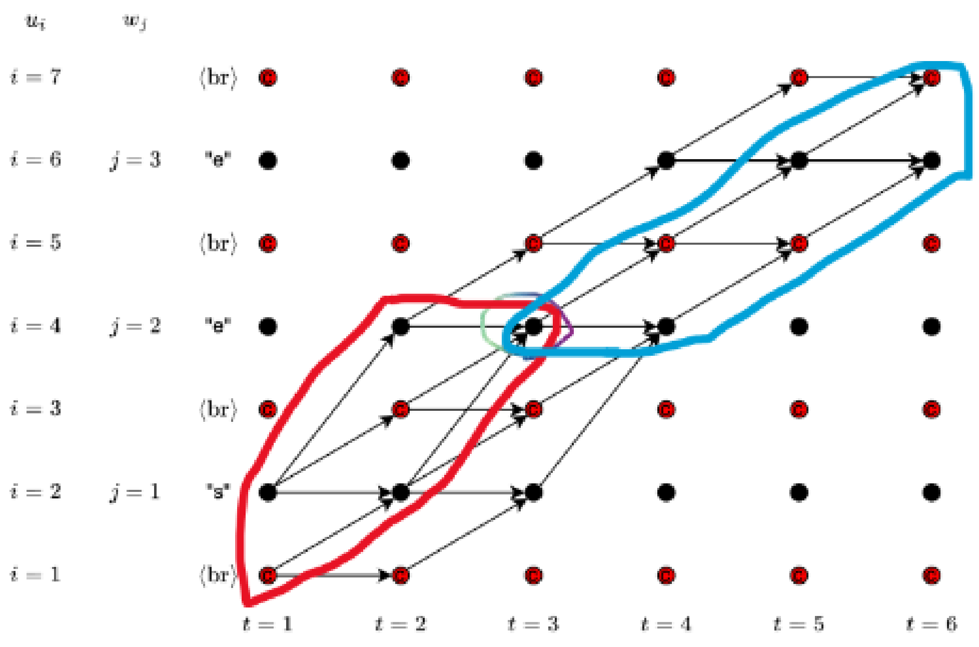
1. assignment: this directory contains 3 files with TODOs: ctc.py, ngram.py, and inference.py. Each file also has local tests in their main functions. You will submit each of these files to Gradescope.
2. utils: this directory contains several files relating to neural network definitions, data loading, and writing decoded outputs to a standard format. You will use decode.py for the beam search portion of this assignment.
3. model: this directory contains a pre-trained neural network. This will be used for the beam search portion of this assignment.
4. data: this directory contains test data. This will be used for the beam search portion of this assignment.

Problem 1

CTC-based Speech Recognition

In this assignment, you'll be building a CTC-based speech recognition system. The provided code template defines an overall training and decoding setup with several key components which are your TODOs. Namely, you will implement 1) the forward and backward computations for CTC, 2) the n-gram language model class, and 3) the greedy and beam search inference algorithms.

CTC Occupation Probability (1 point)



Recall from lectures 10/11 that the CTC occupation probability $\gamma(t, i)$ is defined as follows:

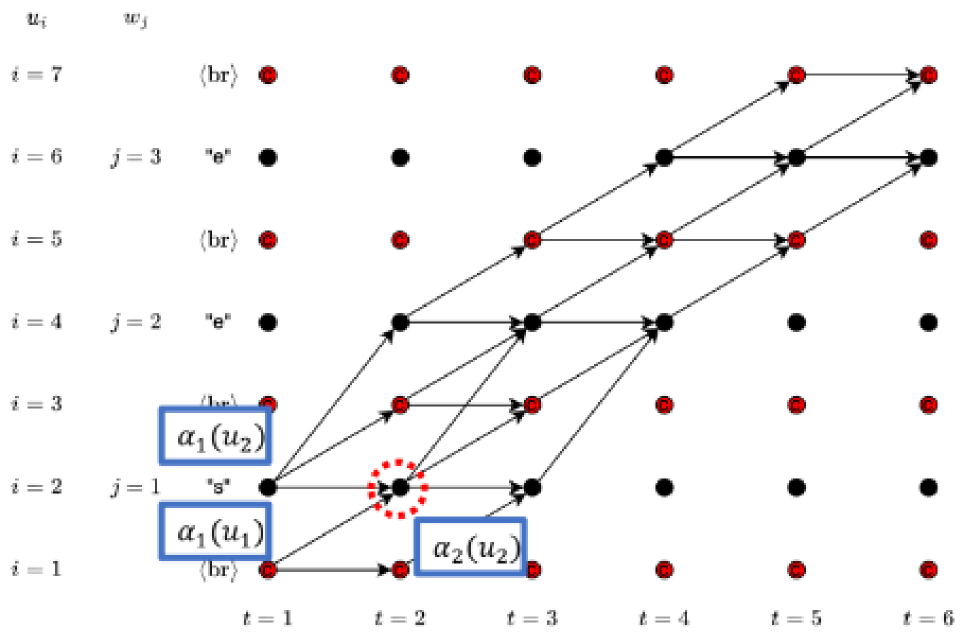
$$\gamma(t,i) = \sum_{Z_{1:t-1}, Z_{t+1,T}} p(Z_{1:t-1}, z_t = u_i, Z_{t+1,T} | O, \Theta), \tag{1}$$

where the summation of all the possible paths from the beginning ($t = 1$) to (t, i), denoted by $\sum_{Z_{1:t-1}}$, is defined as the **CTC forward probability** and the summation of all the possible paths from (t, i) to the end ($t = T$), denoted by $\sum_{Z_{t+1:T}}$, is defined as the **CTC backward probability**.

Task: Complete the `get_occupation_probability` function in ctc.py. This function takes the forward probabilities, `alpha`, and backward probabilities, `beta`, as inputs and returns the occupation probabilities, `gamma`. `alpha`, `beta`, and `gamma` all have the same shape as defined by the CTC trellis. You will implement the computations for `alpha` and `beta` next.

Grading: Submit ctc.py. There are also local tests provided in the main function of ctc.py.

CTC Forward (1 point)



The CTC forward probability $\alpha_t(u_i)$ is defined as:

$$\alpha_t(u_i) = \sum_{Z_{1:t-1} \in f^{-1}(W,T)} p(Z_{1:t-1}, z_t = u_i | O, \Theta), \tag{2}$$

where $f^{-1}(W, T)$ is the set of all paths that generate W. $\alpha_t(u_i)$ should be computed recursively, as in the example in the figure. From $t = 1$ to $t = 2$, $\alpha_2 u_2 = \alpha_1 u_1 y_{u_2 2} + \alpha_1 u_2 y_{u_2 2}$. Note that $y_{u_i t}$ is obtained from the softmax function of a neural network.

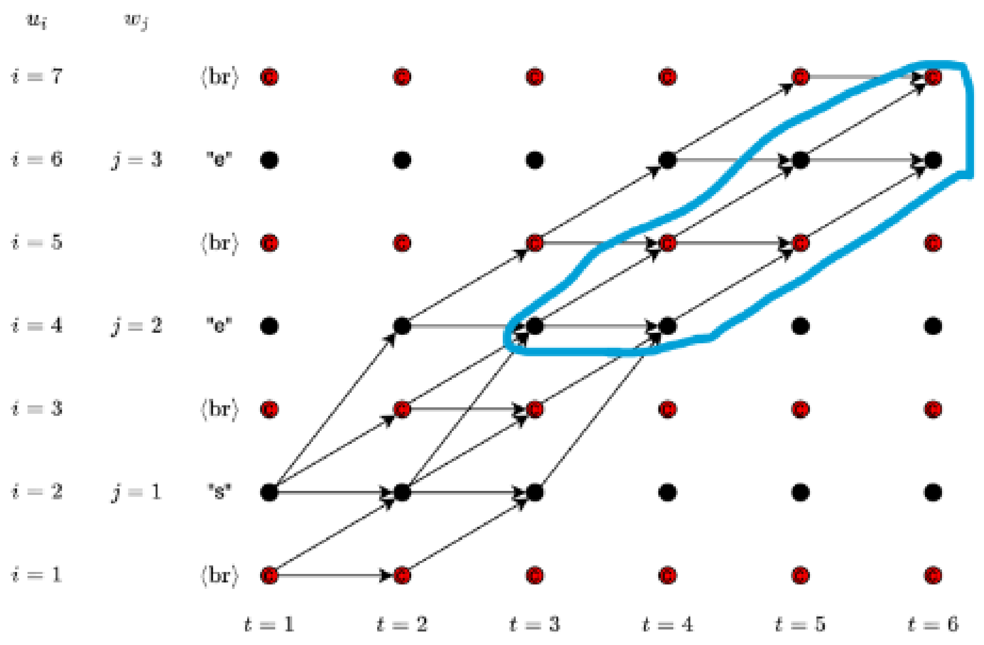
Task: Complete the `get_forward_probability` function in ctc.py. This function takes softmax logits, `logits`, blank-extended symbols, `extended_symbols`, and skip connections, `skip_connections`, as inputs. `logits` are computed by separate (provided) functions handling the neural network forward pass. `logits` provides the values $y_{u_i t}$ and the length of `logits` also defines T, the max length of the x-axis of the CTC trellis. `extended_symbols` and `skip_connections` together define the y-axis of the CTC trellis. `extended_symbols` is the blank-extended symbol list (e.g. [<b>, s, <b>, e, <b>, e, <b>]). `skip_connections` is a corresponding boolean list which indicates when a skip connection is allowed to the position (e.g. [0, 0, 0, 1, 0, 0]; recall that CTC allows skip connections except when there are repeats.

Code hints:

1. Use recursion with memoization. We've provided an empty initialization for `alpha`, a matrix representing the TxI grid). Iterate through t, compute each $\alpha_t(u_i)$ and save the results to `alpha`$[t][i]$
2. Be sure to check `skip_connections` to include all possible paths in your computation
3. Paths can begin with blank or non-blank, so be sure to initialize both `alpha[0][0]` and `alpha[0][1]`

Grading: Submit ctc.py. There are also local tests provided in the main function of ctc.py.

CTC Backward (1 point)



The CTC backward probability $\beta_t(u_i)$ is defined as:

$$\beta_t(u_i) = \sum_{Z_{t+1:T} \in f^{-1}(W,T)} p(Z_{t+1:T} | z_t = u_i, O, \Theta), \tag{3}$$

and is computed recursively in manner similar to the CTC forward probability. Note that at $t = T$, positions with a valid CTC path have $\beta_T(\cdot) = 1$ and all others have $\beta_T(\cdot) = 0$. Generally from $t+1$ to t, $\beta_t(u_i) = \beta_{t+1}(u_i)y_{u_i t+1} + \beta_{t+1}(u_{i+1})y_{u_{i+1} t+1} + \beta_{t+1}(u_{i+2})y_{u_{i+2} t+1}$. The final term is only applied if there is a valid skip connection.

Task: Complete the `get_backward_probability` function in ctc.py. This function takes softmax logits, `logits`, blank-extended symbols, `extended_symbols`, and skip connections, `skip_connections`, as inputs. See the previous section for descriptions of each of these inputs.

Code hints:

1. Use recursion with memoization. We've provided an empty initialization for `beta`, a matrix representing the TxI grid. Iterate through t, compute each $\beta_t(u_i)$ and save the results to `beta`$[t][i]$
2. Be sure to check `skip_connections` to include all possible paths in your computation

3. Paths can end with blank or non-blank, so be sure to initialize both `beta[-1][-1]` and `beta[-1][-2]`

Grading: Submit ctc.py. There are also local tests provided in the main function of ctc.py.

CTC Greedy Decoding (1 point)

Greedy decoding determines the alignment sequence by taking the argmax of the CTC posterior at each time step. After determining the alignment sequence, removing repeats and blanks yields the final output sequence. Hint: the order in which you remove repeats and blanks matters.

Task: Complete the `greedy_decode` function in inference.py. This function takes softmax probabilities, `posterior`, as an input and returns the string output sequence, `output`.

Grading: Submit inference.py. There are also local tests provided in the main function of inference.py.

Kneser-Ney N-gram LM (4 points)

In this problem, you will build a **character-based** Kneser-Ney Language Model.

Kneser-Ney Language model is one of the most commonly used n-gram smoothing models. It belongs to a class called absolute discounting where counts of n-grams are discounted by a fixed number to allocate probability for unseen n-grams. For example, the n-gram probability of character l_i conditioned on previous $n-1$ character l_{i-n+1}^{i-1} can be written in the following form:

$$P_{\text{AbsoluteDiscount}}(l_i|l_{i-n+1}^{i-1}) = \frac{\max(C(l_{i-n+1}^{i-1}, l_i) - d, 0)}{\sum_v C(L_{i-n+1}^{i-1}, v)} + \lambda(l_{i-n+1}^{i-n})P(l_i|l_{i-n+2}^{i-1}) \tag{4}$$

where $C(l_{i-n+1}^{i-1}, l_i)$ is the frequency of n-gram l_{i-n+1}^{i} in the training set, d is a fixed number (0.75 will be used in this assignment), λ is the normalizing factor. The absolute discount probability consists of two parts: the first term on the right-hand side directly measures the probability using the n-gram frequency count, and the second term is to smooth the probability using the lower n-grams. In Kneser-Ney smoothing, the lower n-gram probability $P(l_i|l_{i-n+2}^{i-1})$ is estimated with the unique number of continuation (context) where the n-gram appears.

Bigram Kneser-Ney language model is defined as follows:

$$P_{\text{KN}}(l_2|l_1) = \frac{\max(C(l_1, l_2) - d, 0)}{C(l_1)} + \lambda(l_1)P_{\text{continuation}}(l_2) \tag{5}$$

The probability $P_{\text{continuation}}(l_2)$ is an indicator about how frequently l_2 ends different bigrams

$$P_{\text{continuation}}(l) = \frac{N_{1+}(\star, l)}{N_{1+}(\star, \star)} \tag{6}$$

where $N_{1+}(\star, l)$ is the unique number of bigrams ending with l

$$N_{1+}(\star, l) = |\{v|C(v, l) > 0\}| \tag{7}$$

where $\star$ denotes any character in the vocabulary. Similarly, $N_{1+}(\star, \star)$ is the total number of unique bigrams. $\lambda(l_1)$ is defined to make sure $\sum_{w_2} P_{KN}(l_2|l_1) = 1$, in this case, it is the sum of discounted probability mass.

$$\lambda(l_1) = \frac{dN_{1+}(l_1, \star)}{C(l_1)} \tag{8}$$

Similarly, the trigram Kneser-Ney language model is defined as follows:

$$P_{\text{KN}}(l_3|l_1, l_2) = \frac{\max(C(l_1, l_2, l_3) - d, 0)}{C(l_1, l_2)} + \lambda(l_1, l_2)P_{\text{continuation}}(l_3|l_2) \tag{9}$$

where $P_{\text{continuation}}(l_3|l_2)$ is defined as

$$P_{\text{continuation}}(l_3|l_2) = \frac{\max(N_{1+}(\star, l2, l3) - d, 0)}{N_{1+}(\star, l_2, \star)} + \lambda(l_2)P_{\text{continuation}}(l_3) \tag{10}$$

and $P_{\text{continuation}}(l_3)$ is the same unigram continuation probability we defined above:

$$P_{\text{continuation}}(l_3) = \frac{N_{1+}(\star, l_3)}{N_{1+}(\star, \star)} \tag{11}$$

The normalizing factors $\lambda(l_2), \lambda(l_1, l_2)$ can be derived as

$$\lambda(l_1, l_2) = \frac{dN_{1+}(l_1, l_2, \star)}{C(l_1, l_2)} \tag{12}$$

$$\lambda(l_2) = \frac{dN_{1+}(l_2, \star)}{N_{1+}(\star, l_2, \star)} \tag{13}$$

Note that the trigram is not defined when $C(l_1, l_2)$ is 0. In this case, use $P(l_3|l_2)$ instead. If $C(l_2)$ is 0, then the bigram is also not defined, then use the unigram probability $P(l_3)$ as follows:

$$P(l_3) = \frac{C(l_3)}{C(\star)} \tag{14}$$

where $C(\star)$ is the total number of words in the training set. To prevent the zero probability, you should set $P(l_3)$ to a small number (e.g: $1/|V|$ where $|V|$ is your vocab size) when $C(l_3)$ is zero. Note that in other cases, you should also make sure that the probability is not zero.

By combining trigram and bigram Kneser-Ney models, we can compute the joint probability of a sentence $(l_1, ..., l_n)$ as follows:

$$P(l_1, ..., l_n) = P(l_1|\text{<s>})P(l_2|\text{<s>}, l_1)P(l_3|l_1, l_2) \cdots P(\text{</s>}|l_{n-1}, l_n) \tag{15}$$

where <s> and </s> are special tokens marking the start and end of a sentence.

Task: Complete the functions in ngram.py. Code hints are included in the template. The `fit` function takes a list of sentences which you should process and compute the various counts $C(\cdot)$ and $N_{1+}(\cdot)$. The `unigram_prob`, `bigram_prob`, and `trigram_prob` take a particular n-gram as input for which you are to compute the probability. Finally, the `sentence_prob` function takes a sentence as input for which you are to compute the joint probability.

Code hints:

1. Expect sentences to already be processed such that characters are separated by spaces. Actual spaces have been replaced with "_".
2. Add the start of sentence and end of sentence tokens (e.g. <s> and </s>) to each sentence.

Grading: Submit ngram.py. We will check each of the functions in Gradescope. There are also local tests provided in the main function of ngram.py.

CTC + N-gram Beam Search (2 bonus points)

Finally, we'll put everything together. You'll (optionally) implement beam search with joint CTC and n-gram likelihoods. This algorithm follows the time-synchronous CTC beam search presented in lecture 13.

Algorithm 1: CTC + N-gram Beam Search

```
Define: $\alpha(l,t,u_i) = \sum_{Z_{1:t} \in g^{-1}(l,t)} p(Z_{1:t-1}, z_t = u_i | O, \Theta)$
// Initial end-in-blank and end-in-non-blank likelihoods (for empty prefix at time 0)
1  $\alpha(``", 0, u_i = \text{<b>}) \leftarrow 1$;
2  $\alpha(``", 0, u_i \neq \text{<b>}) \leftarrow 0$;
3  $A_{\text{prev}} \leftarrow \{``"\}$;
4  for $t$ = 1, ..., $T$ do
5      $A_{\text{next}} \leftarrow \{\}$;
6      for $l$ in $A_{\text{prev}}$ do
7          for $c$ in $\mathcal{V}$ do
8              if $c = \text{<b>}$ then
                   // Update end-in-blank likelihood (for prefix $l$ up to time $t$)
9                  $\alpha(l,t,u_i = \text{<b>}) \leftarrow (y_{\text{<b>},t})(\alpha(l,t-1,u_i = \text{<b>}) + \alpha(l,t-1,u_i \neq \text{<b>}))$;
10                 add $l$ to $A_{\text{next}}$;
11             else
12                 $l^+ \leftarrow$ concatenate $l$ and $c$;
13                 if $c = l[-1]$ then
                       /* If the new char is the same as the last, then the prefix may either
                          be extended (with blank in between) or remain the same (repeat).   */
14                     $\alpha(l^+,t,u_i \neq \text{<b>}) \leftarrow (y_{c,t})(\alpha(l,t-1,u_i = \text{<b>}))$;
15                     $\alpha(l,t,u_i \neq \text{<b>}) \leftarrow (y_{c,t})(\alpha(l,t-1,u_i \neq \text{<b>}))$;
16                 else
                       // Update end-in-non-blank likelihood (for prefix $l^+$ up to time $t$)
17                     $\alpha(l^+,t,u_i \neq \text{<b>}) \leftarrow (y_{c,t})(\alpha(l,t-1,u_i = \text{<b>}) + \alpha(l,t-1,u_i \neq \text{<b>}))$;
18                 end
19                 add $l^+$ to $A_{\text{next}}$;
20             end
21         end
22     end
       /* Keep most probable prefixes for next iteration, considering an interpolation of
          LM and CTC scores.  CTC score is obtained from summing the end-in-blank and
          end-in-non-blank likelihoods.  LM score is obtained from an N-gram model.     */
23     for $l$ in $A_{\text{next}}$ do
24         $p(l,t) \leftarrow p_{lm}(l)^{\lambda}(\alpha(l,t,u_i = \text{<b>}) + \alpha(l,t,u_i \neq \text{<b>}))$
25     end
26     $A_{\text{prev}} \leftarrow k$ most probable $l$, defined by $p(l,t)$;
27 end
28 return most probable prefix in$A_{\text{prev}}$
```

Note that this algorithm seeks to estimate the likelihood $\alpha(l, t, u_i)$ of a partial label sequence l. This is analogous to the forward probability during training (except it is not guaranteed that we sum over all paths). The trick to CTC beam search is to estimate this likelihood efficiently via dynamic programming. At a high-level, we iterate over the time dimension of CTC softmax probabilities and build up the likelihoods (considering the possible paths) for possible label sequences. We prune our hypotheses by considering the joint score from CTC and an n-gram model.

Task: Complete the `beam_search` function in inference.py. This function takes CTC softmax probabilities, `posterior`, as an input and returns the string output sequence, `output`. The CTC probabilities come from a neural network (provided) and the n-gram likelihoods come from your implementation in n-gram.py. You will decode a provided test set using decode.py. We have also provided a dev set (with references) so you can compare your greedy WER to your beam search WER.

Example usage of decode.py (You do not have to modify anything inside decode.py):

```
# decode the dev set using greedy
python decode.py --recog_json data/dev/data_char.json --decode_tag dev
# decode the dev set using beam search
python decode.py --recog_json data/dev/data_char.json --decode_tag dev --search_type beam

# decode the test set using greedy. you will not see the WER locally
python decode.py --recog_json data/test/data_char.json --decode_tag test
# decode the test set using beam search
python decode.py --recog_json data/test/data_char.json --decode_tag test --search_type beam
```

Code hints:

1. Test your beam search without the n-gram score portion (just use the CTC score). You should be able to observe a small gain over greedy.
2. You do not need to expand hypotheses by every character in the vocabulary (line 7 of Algorithm 1). You can achieve similar results with faster runtime by expanding only the top 10-15 characters with highest CTC softmax probabilities at the particular time-step.
3. If your implementation is very slow when using n-gram scores, re-consider how you implemented your `fit` function. Are you pre-computing counts?
4. You may choose to tune some hyperparameters: the `--beam_size` and `--lm_weight` arguments can be changed from their default values which are 10 and 0.3 respectively.

Grading: Submit inference.py and your decoded **test** output (e.g. `exp/<model>/<decode>/decoded_hyp.txt`). **Do not submit your dev output by mistake!** We will compute the WER in Gradescope; if your decoding is better than the greedy decoding, you will receive the bonus points.

For reference, the decoded output will look like the following, with different utterances. **Do not manually modify to this file!**

```
1272-128104-0000 MISTER CQLOOR AS THE AP IMPOUSSIL OF THE MIDDLE CLASSES AND WE ARE GLAD
1272-128104-0001 NOR IS MISTER CULTS MANNERLESS INTERESTING THANN HIS METTER
1272-128104-0002 HE TELLS US THAT AT THIS FESTIVEB SEASON OF THE YEAR WITH CHRISTMS
1272-128104-0003 HE HAS G DOBTS WH THEIR SIRFREDERIC LATENS WORK IS READY
```