**ECE C143A/C243A, Fall 2025** **Final project**
Brain–Computer Interfaces Prof. J.C. Kao
University of California, Los Angeles E. Feghhi, TAs S. Athreya & Y. Turali

# Predicting Text from Intracranial Neural Signals

## Introduction

In this project, we will explore the task of predicting text given intracranial neural recordings from a participant with ALS. While we will give a brief background and description of the neural dataset and decoding algorithm, further details can be found in the paper published alongside this dataset: `https://www.nature.com/articles/s41586-023-06377-x` [8].

## Background

The neural dataset used for this project was released as part of the Brain-to-Text Benchmark ’24, a competition organized to help spur machine learning innovation in the field of neural speech decoding. A baseline gated recurrent unit (GRU)-based model was released alongside the dataset. The goal of the benchmark was to improve neural speech decoding performance over this baseline GRU model.

## Neural dataset

The Brain-to-Text ’24 benchmark dataset consists of microelectrode array (MEA) recordings from the ventral premotor cortex (area $6v$) of a single participant with ALS. Data was recorded from two microelectrode arrays with $64$ channels each, a ventral 6v array and a dorsal 6v array. Spike band power and threshold crossings were extracted for each channel, leading to a total of $256$ features. Neural activity was recorded while the patient attempted to speak $10{,}880$ sentences. In each trial, the subject was shown a sentence, and attempted to speak the sentence at the onset of the “go” cue. All decoding analyses are performed on neural activity during the ”go” phase while the participant attempts to speak the sentence. The neural activity is provided in $20$ ms time bins ($50$ Hz resolution), and z-scored within each block ($20$-$50$ sentences). The dataset can be downloaded at the following link: `https://datadryad.org/dataset/doi:10.5061/dryad.x69p8czpq`.

## Train, validation, and test splits

The benchmark provides train, validation, and test splits. There are $8800$ sentences in train, $880$ sentences in validation, and $1200$ sentences in test. Train and validation sentences are recorded across $24$ days (collected over almost $4$ months), and test sentences are recorded on $15$ out of the $24$ days. Performance on the test set can only be calculated by submitting an entry to the competition website, as the text labels for the test set are not provided. A link to the competition website is provided here: `https://eval.ai/web/challenges/challenge-page/2099/overview`.

## Task inputs and outputs

The primary objective is to correctly predict text labels for a given sentence given the neural activity associated with that sentence. The text labels are converted to *phonemes*, since phonemes have a more direct mapping onto the speech sounds the participant is attempting to produce compared to characters. Since there is no ground-truth alignment between phonemes and neural activity, we recommend using the **CTCLoss** [3], a specialized loss function for situations like this.

## Model Evaluation

Your models are to be evaluated using the **Phoneme Error Rate (PER)** metric. The PER computes:

$$\text{PER} = \frac{S + D + I}{N},$$

where $S$ is the number of substitutions, $D$ is the number of deletions, $I$ is the number of insertions, and $N$ is the total number of characters. It is also possible to compute **Word Error Rate (WER)** by converting the decoded phonemes into words, for instance by using a weighted finite state transducer (WFST)-based decoder [6]. Due to the computational costs of using a WFST decoder, we will not require this for this project. However, interested students may explore ways to compute WER. Since evaluation on the test set requires converting decoded text into words, the main focus of this project will be to improve the PER on the validation set.

## CTC Loss

Proposed in 2006 [3], the **Connectionist Temporal Classification** (CTC) loss aims to address classification under settings with **unsegmented sequential data**. This refers to situations like ours where the labeled sequence consists of phonemes with no ground-truth timing. The CTC loss is already implemented in the codebase, however, we encourage you to read more indepth about it for a fuller understanding (e.g., `https://distill.pub/2017/ctc/`).

## Baseline Gated Recurrent Unit Model

The benchmark neural dataset is accompanied by a baseline gated recurrent unit (GRU)-based model. We provide a description of this baseline model here. This model first passes the input features (spike band power and threshold crossings) through a day-specific linear layer followed by a softsign non-linearity, which serves to account for day-specific differences in neural activity. At each step, the GRU receives a vector of these day-transformed inputs with dimension $F \cdot T_{in}$, where $F$ is the number of features ($256$) and $T_{in}$ (kernel length) is the number of neural time bins. The baseline model consists of $5$ GRU layers. The hidden state of the final layer is passed through a linear layer to produce logits over phonemes, the CTC blank token, and a silence token. Logits are output every $T_{out}$ (stride length) steps. For optimal performance, the baseline algorithm sets $T_{in} = 32$ and $T_{out} = 4$, resulting in an $87.5\%$ overlap between consecutive inputs. The model is trained with the Adam optimizer and CTC Loss. During training, white noise and baseline shift augmentations are added to the neural activity, followed by causal Gaussian smoothing. Training is performed for $10{,}000$ batches.

## Setting up the code environment

The GitHub repository containing a notebook to format the dataset and the necessary scripts to run the baseline model can be found at `https://github.com/cffan/neural_seq_decoder`. We provide a set of steps below to properly setup the code base.

1. Clone the github repository.
2. Create a virtual environment. We recommend using the `uv` package manager, and installation instructions can be found here: `https://docs.astral.sh/uv/getting-started/installation/#standalone-installer`
3. Create a virtual environment by running: `uv venv -p 3.9`.
4. Install the necessary packages for the codebase by running `uv pip install -e .`
5. You can activate the virtual env by typing `source .venv/bin/activate`. Alternatively, write `uv` before running pip or python commands to activate the environment.

## Formatting the dataset

The codebase provides a notebook to format the dataset, which is located in the notebooks folder. In order to easily run the notebook, we recommend first running `uv pip install ipykernel` and then activating the `.venv` inside the notebook. Running the notebook simply requires changing the `dataDir` variable to point to the location where you downloaded the dataset from Dryad. Additionally, you must change the path where the dataset is saved.

## Training and evaluating a baseline model

Once the dataset is successfully saved, you can begin training the baseline GRU model. To train the model, first open the `train_model.py` file located inside the `scripts` folder. Modify the `datasetPath` and `outputDir` keys in the `args` dictionary to match the location the data is stored in and the folder you want to save the model outputs in, respectively. For this project, we will focus on models that decode speech in real-time instead of waiting for the entire trial to finish. As such, also modify the `bidirectional` key by setting it to `False`. To begin training, run `uv python train_model.py` while located inside the `scripts` folder. For reference, training takes roughly 30 minutes on an Nvidia GeForce RTX 3090 GPU. (If for some reason you desired to use the CPU, which is slower, navigate to the following python file `src/neural_decoder/neural_decoder_trainer.py`, and modifying the `device` variable from `cuda` to `cpu`.) While training, the script will output the validation CTC loss and PER (written as CER in the script) for every 100 batches. The model should achieve a PER of $\sim 22\%$ after training has completed. We provide a description for each of the important arguments in the args dictionary below.

- `outputDir`: location to save the model outputs.
- `datasetPath`: location where the data is saved, set in `formatCompetitionData.ipynb`.
- `batchSize`: number of training examples to include per batch.

- `lrStart`: beginning learning rate.
- `lrEnd`: final learning rate. Set to a value lower than `lrStart` to have the learning rate linearly decay, otherwise set to `lrStart` to have no learning rate decay.
- `nUnits`: hidden dimensionality of the GRU.
- `nBatch`: number of batches to train for.
- `nLayers`: number of GRU layers.
- `seed`: random seed, used for reproducibility.
- `nClasses`: number of output classes, not including the CTC blank token.
- `nInputFeatures`: number of neural features (spike band power and threshold crossings).
- `dropout`: dropout percentage used for GRU layers.
- `whiteNoiseSD`: amount of white noise augmentation to add to neural data during training.
- `constantOffsetSD`: amount of baseline shift augmentation to add to neural data during training.
- `gaussianSmoothWidth`: convolves the neural data with a Gaussian kernel with the specified width.
- `strideLen`: number of neural time bins the input is shifted forward at each timestep. This controls how often the GRU makes an output.
- `kernelLen`: number of neural time bins fed to the GRU at each timestep.
- `bidirectional`: if True, runs the GRU in bidirectional mode.
- `l2 decay`: amount of L2 regularization that is applied.

## Project Directions

The goal of this project is to improve upon the baseline GRU model. The most straightforward way to do this is to improve upon the validation PER achieved by the baseline model. However, you may also optimize the baseline GRU model based on other criteria, for instance reducing the model parameter count or training time while maintaining competitive performance. We suggest some avenues for exploration below, and cite relevant papers in our suggestions. It may be helpful to skim through these cited papers to get a better idea of what other researchers have done to improve performance. Note that the difficulty of the topics listed below ranges, and we recommend students pick a topic based on their prior experience with these methods.

1. `Model architecture`: Read and understand the code in the python script defining the model architecture, located in `src/neural_decoder/model.py`. Experiment with modifying the existing model architecture or using another model architecture. For instance, the ”Linderman Lab” entry in the benchmark summary report [9] found that incorporating a stack of linear,

layer normalization, and dropout layers after the bidirectional GRU layers helped improved performance. You may also experiment with adding layer normalization layers before the GRU layers and see if it removes the need for day-specific parameters, as done in [2] with a Transformer architecture.

2. `Augmentations`: Experiment with different values for the current augmentations (white noise and baseline shift), or introduce new augmentation methods. For instance, [1] increase the white noise augmentation from $0.8$ to $1$ on a more recently collected neural dataset, and [2] introduce a time-masking augmentation which masks contiguous temporal chunks of neural activity during training. Interesting augmentations to try could be masking entire features (feature-masking), or applying feature-masking and time-masking jointly.

3. `Optimizer`: Experiment with using different optimizers than Adam. For instance, AdamW was used in both [1] and [2]. Additionally, you may notice the `epsilon` value is relatively large in the `src/neural_decoder/neural_decoder_trainer.py` file relative to the Pytorch default. You may try exploring the impact of this.

4. `Data transformations`: Neural data was log-transformed in [2] before z-score normalization, and this was found to help performance. You can experiment with log transforming the neural data or other transformations to the neural data before training.

5. `Output representation`: The current baseline decodes each phoneme individually. However, phonemes are known to be context dependent. In other words, the articulatory gestures required to produce a given phoneme may change depending on the surrounding phonemes. [5] takes this into account by decoding diphones, which represents the transitions between phonemes. You can experiment with using context-dependent output representations such as diphones, or other output representations.

6. `Loss function`: The baseline algorithm uses the CTC loss function. Since the publication of the CTC loss, there have been several attempts to directly improve upon it ([10], [4], [7]). You may experiment with modifying the CTC loss.

Feel free to innovate beyond the components above to earn points for creativity and insight. Extra insight points may also be rewarded for explaining how these different approaches result in better or worse performance.

## Project Logistics

### Submittables

Each group should submit a writeup of their project work, exceeding no more than 7 pages. References are excluded from this page count. It is fine to be below the page limit; this is the *maximum*. We will also ask you to submit your code, so that we can validate your results. If you have a project where you cannot submit your code, please notify us so we can proceed accordingly.

The writeup must adhere to the following template: `https://media.neurips.cc/Conferences/NeurIPS2024/Styles.zip` – so that we can judge all writeups in the same manner without having to worry about different font sizes, etc. To remove the line numbers, modify the following line at the beginning of the `.tex` file to include the `final` option: `\usepackage[final]{neurips_2024}`.

## Writeup

In the writeup, there should be the following sections:

1. **Abstract** – A brief description of what you did in the project and the results observed.
2. **Introduction** – Use the introduction to set up and motivate the question and techniques you pursued.
3. **Methods** – State the methods of your project (such as the architectures, data preprocessing, or other techniques used).
4. **Results** – State the results of your experiments.
5. **Discussion** – Discuss insights gained from your project, e.g., what resulted in good performance, and any hypotheses for why this might be the case.
6. **References** – List references used in your writeup.

## Grading

Here we outline the criterion by which we will grade the project. Note that some projects will be more creative than others; some projects will achieve higher performance than others. We will provide room for extraordinary work in one category to compensate for deficiencies in another category. These are the general areas we will look into. Concretely, the final project will be graded on a scale of 20 points, but each section is assigned points so that the sum total can exceed 20 points. Your final project score will be capped at 20 points. You should aim to do a good job in all areas.

1. **Creativity (7 points)**
   - How creative and/or diverse is the approach taken by the student(s)?
   - Do the student(s) implement and try various modifications?
2. **Insight (7 points)**
   - Does the project reveal some insight about why approaches work or did not work?
   - Is there reasonable insight, explanation, or intuition into the results? (i.e. you should not just blindly apply different modifications to a problem and compare them.)
3. **Performance (6 points)**
   - Does the project achieve relatively good performance?
4. **Write-up (4 points)**
   - Are the approach, insight, and results clearly presented and explained?

*Dissemination of work is an important component to any project.*

Finally, if any students achieve excellent PER and would like to consider extending to calculate WER through a language model, potentially leading to a new research paper, you are welcome to reach out to `efeghhi@gmail.com` and CC Prof. Kao at `kao@seas.ucla.edu`. We do not guarantee we have the capacity to help, but for particularly promising approaches where you would like our input and help, we will consider it.

## References

[1] Nicholas S Card, Maitreyee Wairagkar, Carrina Iacobacci, Xianda Hou, Tyler Singer-Clark, Francis R Willett, Erin M Kunz, Chaofei Fan, Maryam Vahdati Nia, Darrel R Deo, Aparna Srinivasan, Eun Young Choi, Matthew F Glasser, Leigh R Hochberg, Jaimie M Henderson, Kiarash Shahlaie, Sergey D Stavisky, and David M Brandman. An accurate and rapidly calibrating speech neuroprosthesis. *N. Engl. J. Med.*, 391(7):609–618, August 2024.

[2] Ebrahim Feghhi, Shreyas Kaasyap, Nima Hadidi, and Jonathan C Kao. Time-masked transformers with lightweight test-time adaptation for neural speech decoding. *arXiv [cs.HC]*, July 2025.

[3] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural 'networks. volume 2006, pages 369–376, 01 2006.

[4] Jaesong Lee and Shinji Watanabe. Intermediate loss regularization for CTC-based speech recognition. *arXiv [eess.AS]*, February 2021.

[5] Jingyuan Li, Trung Le, Chaofei Fan, Mingfei Chen, and Eli Shlizerman. Brain-to-text decoding with context-aware neural representations and large language models. *arXiv [eess.SP]*, November 2024.

[6] Mehryar Mohri, Fernando Pereira, and Michael Riley. Speech recognition with weighted finite-state transducers. In *Springer Handbook of Speech Processing*, pages 559–584. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[7] Jumon Nozaki and Tatsuya Komatsu. Relaxing the conditional independence assumption of CTC-based ASR by conditioning on intermediate predictions. In *Interspeech 2021*, ISCA, August 2021. ISCA.

[8] Francis R Willett, Erin M Kunz, Chaofei Fan, Donald T Avansino, Guy H Wilson, Eun Young Choi, Foram Kamdar, Matthew F Glasser, Leigh R Hochberg, Shaul Druckmann, Krishna V Shenoy, and Jaimie M Henderson. A high-performance speech neuroprosthesis. *Nature*, 620(7976):1031–1036, August 2023.

[9] Francis R Willett, Jingyuan Li, Trung Le, Chaofei Fan, Mingfei Chen, Eli Shlizerman, Yue Chen, Xin Zheng, Tatsuo S Okubo, Tyler Benster, Hyun Dong Lee, Maxwell Kounga, E Kelly Buchanan, David Zoltowski, Scott W Linderman, and Jaimie M Henderson. Brain-to-text benchmark '24: Lessons learned. *arXiv [cs.CL]*, December 2024.

[10] Zengwei Yao, Wei Kang, Xiaoyu Yang, Fangjun Kuang, Liyong Guo, Han Zhu, Zengrui Jin, Zhaoqing Li, Long Lin, and Daniel Povey. CR-CTC: Consistency regularization on CTC for improved speech recognition. *arXiv [eess.AS]*, October 2024.