Speech reproduction from latent representations of neuronal spiking activity in the Motor Cortex

December 16, 2024

Abstract

This project explores phoneme decoding from latent feature representations of high-dimensional neural spiking activity in the motor cortical areas of the brain. I experimented with and discovered which latent feature extraction techniques are appropriate for encoding of neuronal activity, as well as aimed to see whether unsupervised latent feature extraction can still retain behaviorally relevant information. There seems to be feasibility in decoding phonemes even when the encoder-decoder pipeline is separated, where the encoder learns latent features completely unsupervised to the labels themselves. Although my techniques and accuracies using latent feature representations as input to decoder models do not supersede the baseline provided in the reference study, we can still see that there is efficacy in such unsupervised encoding approaches, reaching 50% character error rate (CER). In conjunction with a language model, speech decoding can reach high levels of accuracies.

I. Introduction

The relationship between orofacial movement and speech, particularly phoneme production, has been observed in the motor cortical areas of the brain. Neural activity recorded with high temporal resolution, coupled with preprocessing methods that highlight the innate dynamical structure of the spiking activity of this localized area of the brain, has proven to encode information about phonemes. Researchers have been able to accurately decode this information with advanced deep learning algorithms.

The study that will be referenced in this paper is one consisting of trials of a patient with Lateral Sclerosis, a form of paralysis that restricts their ability to communicate [1]. In each trial, the patient is shown a sentence, and their spiking activity is recorded as they attempt to reproduce the sentence. The researchers involved in this experiment mainly used RNNs as the deep learning architecture of choice, as these models lend themselves well to decoding sequential data, such as time series of spiking activity. Their proposed methods have been successful, reaching a 23.8% error rate in decoding speech with a 150,000 word vocabulary [1]. The benchmarks provided in this paper are reassuring and demonstrate the efficacy of deep learning methods in extracting useful information from high-dimensional data at fine temporal scales.

The direction in which this paper seeks to explore is to gain a more nuanced understanding of the underlying dynamics of motor cortical neurons themselves. Spiking activity, especially, provides an opportunity to compare different state-space models, from linear-Gaussian models to more generalized linear models, to deep learning models where the state-space model structure is not defined. This paper will compare different modeling strategies for encoding of neural activity,

as well as consider the viability of reducing the complexity of deep learning decoders due to effective latent feature representations as input.

II. Encoders

I start by formalizing my different approaches in inferring the dynamics of neural activity. These methods can be separated into the following stages, where each stage provides a host of design choices.

- Choosing encoder models
- Fitting these models

Choosing Encoder Models: I split encoder models into two different types for this paper: 1. *Dynamical state-space models* and 2. *deep learning encoder models* where the model characteristics are not assumed.

1. **State-space models**: Due to the discrete nature of single-neuron spiking activity, where its dynamics is best described by discrete events in time, they are well modeled via point process measures. However, the dataset provided by the researchers has been Gaussian smoothed with 50 ms windows, which changes its innate structure. To this end, linear-Gaussian state-space modeling may perform better than point-process modeling. I will validate this hypothesis later in the paper, using measures such as R2 scores and neural self-prediction CC. In the Poisson model, I use a generalized linear model that basically differs from linear models in two major ways. First, there is a non-scalar link function that relates the linear combination of predictors with the target value. Second, the target value in a Poisson model is the instantaneous firing rate, rather than just the predicted value of the signal. Formally,

   $$\lambda(t) = f(\beta_0 + \beta X_i)$$

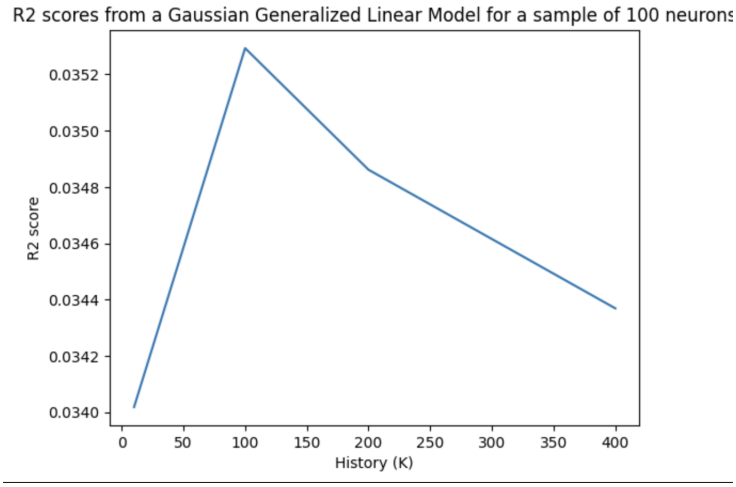
   where f is a non-linear function such as a log-link, and λ is a time-varying prediction of the instantaneous firing rates of neurons.

2. **Deep Learning encoder models**: In the given speech study, researchers have achieved high levels of accuracy using just the spiking activity and a gated recurrent unit (GRU) decoder model. There is no explicit separation between the encoder and decoder in their methods. One potential consequence is that the neural network has low explainability, and it is hard to reason about exactly which neural features are contribute to the phoneme classification.

Using encoder models that extract low-dimensional feature representations that not only preserve the dynamics of neural activity but also illuminate properties of behavior in lower-dimensionality space can be useful especially in terms of explainability of the models. I will compare two different unsupervised latent state learning algorithms, namely EM and SID, both from Professor Shanechi's lab.

Fitting a state space model: The algorithms that are generally used in point process and linear gaussian state space modeling include Expectation Maximization and Maximum Likelihood Estimation. As an example, I show the results of fitting separate generalized linear models for 100 neurons in the 6V array in the ventral premotor cortex across one day's neural reocrdings, where the scoring measure is the self-prediction R^2 score. I want to emphasize that each GLM is trained

separately, and only a neuron's own history comprises the features in its generalized linear model. The scores are extremely low, and suggest that a neuron's firing rate depends not only on its own history, but on the neurons in its vicinity.

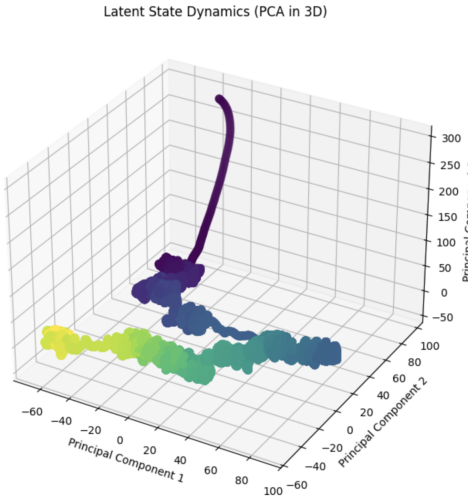


These results demonstrate that accurate modeling of spiking activity in the brain depends on interactions between neurons.

To explore this, I looked at unsupervised neural dynamical modeling of the brain that learns from ensembles of neurons and extracts latent states, which are assumed to be the lower-dimensional driving force of dynamics and behavior. For my case, I considered one day's worth of data; specifically, the data comprises of 128 neurons in the 6v ventral premotor cortex, and the spiking activity is obstained with threshold crossing measures. I then evaluated **self-prediction CC** using these neural dynamical models. In expectation maximization, the core step is finding an x such that U(x, x') $\geq$ U(x', x') for the current estimate of the parameters x', where

$$U(x, x') = \mathbb{E}[log(p_z(z; x))|y = y; x']$$

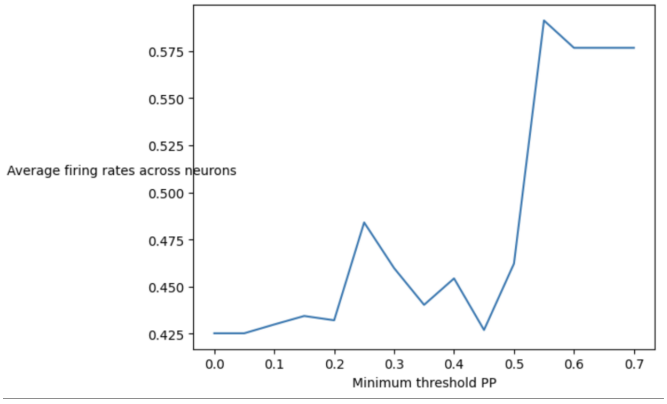
. The issue with generalized EM is that iterations can take long periods of time to converge due to maximization steps that only marginally improve U(x, x') over U(x', x'). In my case, I had to stop EM after 30 iterations due to the computational effort and time. Below I show the evolution of 16-dimensional smoothed latent states learned from EM where the underlying model is a point process model. The neural activity is from just one day of trials. To construct this visualization, I projected the latent states to their top three principal components, and I plot them in time.



Some interesting aspects to note are that although there is a marked gradual evolution of the states in time, local dynamics seem to have high frequency components. Using these latent states and the GLM parameters obtained from Professor Shanechi's EM algorithm, I performed self-prediction,

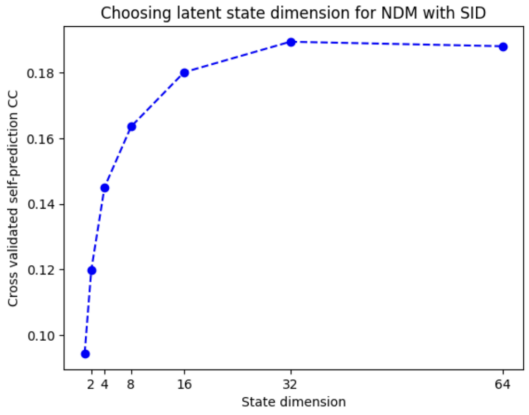
$$\hat{N(t)} = exp(\begin{bmatrix} — & \theta_1^T & — \\ — & \theta_2^T & — \\ — & \dots & — \\ — & \theta_d^T & — \end{bmatrix} \begin{bmatrix} 1 \\ x(t)_1 \\ \vdots \\ x(t)_d \end{bmatrix})\Delta$$

where d is the dimension of the latent states. I computed the self-prediction accuracy of the learned dynamical model by comparing $\hat{N(t)}$ with the true spike count N(t) and calculating PP = 2 * AUC ROC - 1. Using this calculation, I get a mean predictive power (PP) of 0.205, which suggests that the model has, in fact, learned some dynamics. Continuing with my analysis, I was interested in seeing whether there was a relationship between neurons that fired more frequently and their dynamical model's predictive power. In the graph shown below, note that as the threshold for minimum predictive power increases, the average firing rate across neurons increases.



This seems to be an intuitive result, since the dynamics for neurons that fire extremely often can be more easily learned, and therefore the model is able to predict such neurons' one-step-ahead behavior with greater accuracy.

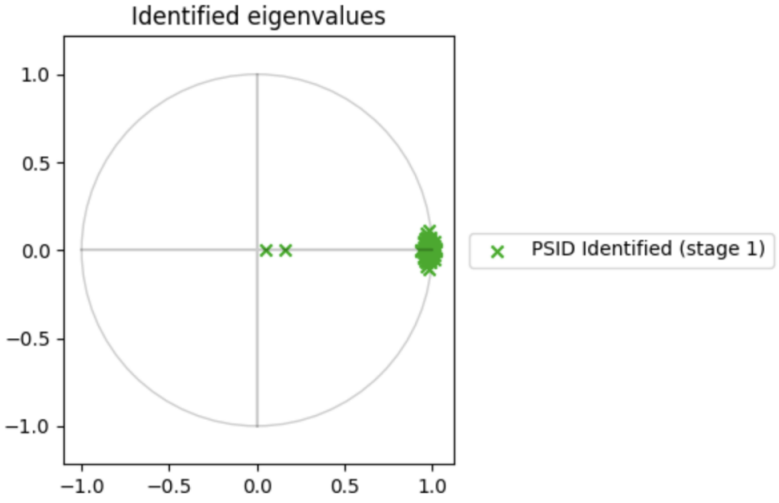
A powerful and efficient technique for neural dynamical modeling is subspace identification (SID), which makes use of linear projections to extract a lower-dimensional feature space that describes the underlying dynamics of neural activity. Due to its linear and noniterative approach, this analytic method runs orders of magnitude faster than EM. To take advantage of the efficiency of this algorithm, I performed SID with different latent state dimensionalities, and cross-validated the self-prediction CC over an entire day of data.



As we increase the state dimension beyond 32, the model does not gain much in self-prediction. In the results section, I will demonstrate the accuracy of decoders with different input dimensionalities.

A surprising result when using SID to extract higher-dimensional latent states is in the eigenvalue distribution of the state-space matrix A. For dimensions 2 through 32, the eigenvalues are clustered around $0.99 \pm \alpha j$ where $\alpha \leq 0.2$. However, when performing SID with 64 dimensions, there are additional eigenvalues with real parts closer to 0.

- 0.04586511+0i
- 0.16758648+0i



An interesting question to consider is whether these additional modes are predictive of phonemes, or whether they just represent high-frequency noise. For future work, I hope to investigate this result [2].

III. Phoneme Representations in Neural Activity

Finding an appropriate context length for input to a phoneme classifier requires an understanding of phoneme representations of the brain. The relevant information to consider when designing inputs to a neural network include: where in the brain are these representations generally encoded? How long are these representations reflected in neural signals? How invariant are these neural representations in regards to their positions in words and sentences? Thanks to extensive research by the authors of the speech study, there is overwhelming evidence that for the given participant, neurons in the 6V (ventral premotor cortex) area explain a large portion of variance in phoneme production; additionally, they show that Area 44 (in Broca's area) has little information regarding phoneme representation [1]. For this report, I only focus on neurons in Area 6V.

II. Decoders

Before presenting the decoders used in this project for phoneme classification, I will explain the data collection format, which motivates some of the decisions made for decoder models and also restricts certain types of techniques from being employed. The patient's neural activity was recorded in sequences of days, where each day had sets of blocks of trials. There may be an unknown waiting period between blocks, which can potentially introduce non-stationarity in the data. Each trial consists of a patient speaking a variable-length sentence as the words appear on the screen. The total time the sentence is spoken is not fixed, nor is the number of words or phonemes in the sentence. Although the dataset and the neural recordings themselves are of high quality, there are some challenges in analysis of the data: decoder models must be able to capture the temporal relationships between sequential neural features, and predictions must be handled in a way that allows for specific alignments between neural data and phoneme labels.

The authors suggest CTC loss, which is what I used as well in my experiments, as it addresses the alignment problem extremely well. Essentially, the CTC loss finds a probability distribution $\{\mathbf{y_{(s,t)}}\}$ for the target sequence, where probable sequences under this distribution represent time series that align well with their label (sequence of phonemes). The essential idea is that it uses both forward-iterating and backward-iterating probabilities to compute the negative sum log likelihood of the current sequence of phonemes, under the target distribution. This loss function is:

$$l = -\sum_{i=1}^{T-1} logP_{(s,t)}$$

where $P_{(s,t)}$ is the probability of the reaching position s in the target sequence at time t [3]. High values of the loss function show that the sequence of neural features do not seem to correlate with the label well, in any form of dynamic alignment. The probability distribution is learned by adjusting the parameters of the predictive model such that its outputs are better aligned with the labels $\{\mathbf{y_{(s,t)}}\}$.

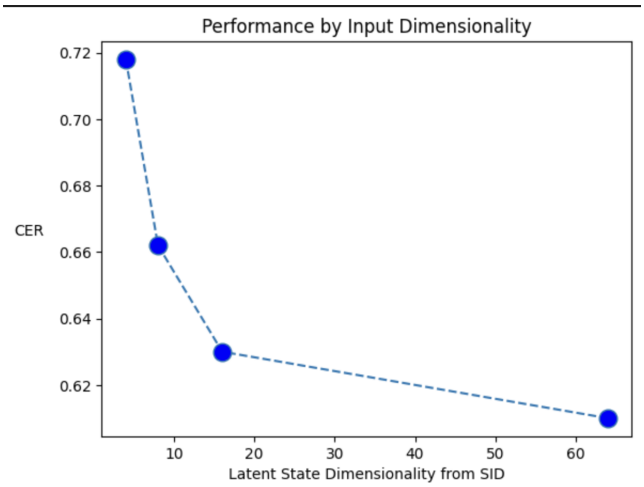
Because we need dynamical alignment with the labels, simple classifiers such as SVM and Naive Bayes are not appropriate for this type of data. So, it is only feasible to train models where there is iterative learning, via computing gradients of a loss function.

To see whether there is some sort of relationship between time windows of evolving latent states and phonemes, I took inspiration from parameters for RNNs. Using concepts such as kernel dimension and stride, I thought to define fixed-length input features that encode temporal evolution

and use them as input to a feed-forward neural network. Functions such as **torch.unfold** help with segmenting 3D B x C x T (Batch x Channel x Time) tensors and splitting up the time dimension so that each sample in the batch contains time segments of concatenated latent vectors. The result is a tensor of the form: (Batch x Time segments x Feature vectors). My approach was to flatten the first two dimensions and treat each time segment within a batch sample independently where each feature vector is a set of concatenated latent states within that block. I emphasize that since the **stride** << **kernel**, there is significant overlap of time between samples.

Using a baseline kernel length of 32 and stride of 4, as this is what the authors implemented for RNNs, I trained a 2-layer feed-forward neural network to attempt classification of 41 phonemes (including the "blank" phoneme). Due to a lack of computational resources, only one day of trials was used for training in my experiments. I first tried EM with 16-dimensional latent states, but switched over to Subspace Identification due to suboptimal results. Using these latent states, the feed-forward neural network was able to get around **70% CER** (cross-validated average).

However, I was able to get better accuracies with SID at lower dimensionalities, so I used that method for the remainder of my analysis.



The x-axis in the figure shown above is the latent state dimensionality obtained with SID. The y-axis in the figure shown above is the character error rate (CER), obtained with CTC loss measures. I took advantage of the fact that there are separate blocks of trials throughout the day, and performed cross validation with leave-one-block-out CV. Since there were 14 blocks of trials, the reported CERs are averaged on 14-fold CV. I want to emphasize that these accuracies may be heavily dependent on the type of normalization performed on the data. The choices are manifold:

- Within-sentence normalization of latent states
- Within-block normalization of latent states
- Batch Normalization during training
- Layer Normalization during training

The CERs reported above are a result of within-sentence normalization of latent states, but this raises some concerns. This assumes non-stationarity from sentence to sentence, which can be problematic since feature vectors across sentences describing the same phoneme are not necessarily

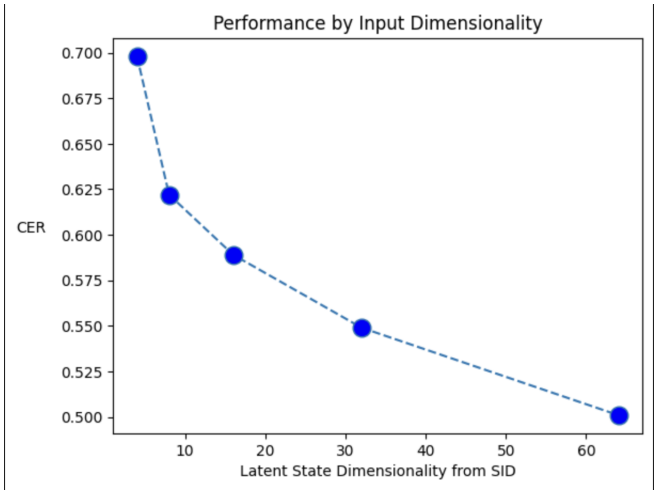
Normalization	Dimensionality	CER
Across-sentence	4	0.72
Across-sentence	8	0.67
Across-sentence	16	0.606
Across-sentence	64	0.56
Within-sentence	4	0.72
Within-sentence	8	0.66
Within-sentence	16	0.63
Within-sentence	64	0.61

Table 1 Comparison between Normalization Methods

different. However, by only doing within-sentence normalization we may be introducing divergence between similar latent representations due to differing mean and variance metrics across sentences. To mitigate this concern, I also attempt within-block normalization of latent states to preserve latent state representations across sentences.

As can be seen in the table above, as the dimensionality increases, across-sentence normalization does seem to help in decoding accuracy. This validates my claim that there is not much nonstationarity from sentence to sentence.

For my other experiment, I modified the researchers' RNN to accept latent state features as input rather than the discrete threshold crossing rates. To make this modification, the size of the network had to be substantially reduced, since the input feature dimensionality is much smaller. The number of layers was reduced from 5 to 3, and the number of neurons in the hidden layers was also reduced. Using the same CTC loss function, but this time learning the data sequentially rather than in a feed-forward manner, these are the results:



Although I was not able to produce CER rates lower than the researchers' original model where the input consists of the entire spiking activity (cross-validated average CER Rates of 0.4 to 0.45 for this day of trials), it is encouraging to note that the model is able to achieve 0.5 CER rates on classification of 40 phonemes. Additionally, the computation time to train an RNN with latent inputs instead of all the spiking activity was reduced by a large factor, by a factor of about 5.

IV. Discussion

In this project, I experimented with various dimensionality reduction techniques, particularly unsupervised methods of extracting latent feature representations of high-dimensional neural spiking activity in the motor cortical areas of the brain. It seemed that Expectation Maximization did not produce as behaviorally predictive features as Subspace Identification. Using SID, with different latent state dimensionalities that reduce the neural feature space by at least a factor of 2, we are able to get 56 % CER with a multi-layer perceptron, and around 50 % CER with an RNN. I experimented with different normalization techniques that preserved latent representations across sentences, but not across blocks, as to preserve latent feature representations across sentences but also account for the possible presence of nonstationarities in neural activity.

For future work, I want to better interpret the predictive modes of the dynamical state-space model returned from unsupervised methods such as EM or PSID. Additionally, I am interested in exploring Convolutional Neural Networks and how they can be used as input layers for feature extraction, and then use their output as input to an RNN so the model has access to the most behaviorally relevant features for phoneme classification.

Code

My code as well as modifications to the code provided by the researchers of the study can be accessed below. **Please note that I used their RNN with some modifications, as well as their pipeline for training with CTC loss. All other notebooks are my own. The RNN code is available here: Neural Sequence Decoder, `https://github.com/cffan/neural_seq_decoder/blob/master/README.md`**

- **Code for formatting data**

```
sessionNames = ['t12.2022.04.28',  't12.2022.05.26',  't12.2022.06.21',  't12.2
't12.2022.05.05',  't12.2022.06.02',  't12.2022.06.23',  't12.2022.07.27',  't1
't12.2022.05.17',  't12.2022.06.07',  't12.2022.06.28',  't12.2022.07.29',  't1
't12.2022.05.19',  't12.2022.06.14',  't12.2022.07.05',  't12.2022.08.02',  't1
't12.2022.05.24',  't12.2022.06.16',  't12.2022.07.14',  't12.2022.08.11']
sessionNames.sort()
import re
from g2p_en import G2p
import numpy as np

g2p = G2p()
PHONE_DEF = [
    'AA', 'AE', 'AH', 'AO', 'AW',
    'AY', 'B',  'CH', 'D', 'DH',
    'EH', 'ER', 'EY', 'F', 'G',
    'HH', 'IH', 'IY', 'JH', 'K',
    'L', 'M', 'N', 'NG', 'OW',
    'OY', 'P', 'R', 'S', 'SH',
    'T', 'TH', 'UH', 'UW', 'V',
```

```
    'W', 'Y', 'Z', 'ZH'
]
PHONE_DEF_SIL = PHONE_DEF + ['SIL']

def phoneToId(p):
    return PHONE_DEF_SIL.index(p)
import scipy

def loadFeaturesAndNormalize(sessionPath):

    dat = scipy.io.loadmat(sessionPath)

    input_features = []
    transcriptions = []
    frame_lens = []
    block_means = []
    block_stds = []
    n_trials = dat['sentenceText'].shape[0]

    #collect area 6v tx1 and spikePow features
    for i in range(n_trials):
        #get time series of TX and spike power for this trial
        #first 128 columns = area 6v only
        features = np.concatenate([dat['tx1'][0,i][:,0:128], dat['spikePow'][0,

        sentence_len = features.shape[0]
        sentence = dat['sentenceText'][i].strip()

        input_features.append(features)
        transcriptions.append(sentence)
        frame_lens.append(sentence_len)

    #block-wise feature normalization
    blockNums = np.squeeze(dat['blockIdx'])
    blockList = np.unique(blockNums)
    blocks = []
    for b in range(len(blockList)):
        sentIdx = np.argwhere(blockNums==blockList[b])
        sentIdx = sentIdx[:,0].astype(np.int32)
        blocks.append(sentIdx)

    for b in range(len(blocks)):
        feats = np.concatenate(input_features[blocks[b][0]:(blocks[b][-1]+1)],
        feats_mean = np.mean(feats, axis=0, keepdims=True)
        feats_std = np.std(feats, axis=0, keepdims=True)
        for i in blocks[b]:
```

```
            input_features[i] = (input_features[i] - feats_mean) / (feats_std +

    #convert to tfRecord file
    session_data = {
        'inputFeatures': input_features,
        'transcriptions': transcriptions,
        'frameLens': frame_lens,
        'blockIdx': blockNums
    }

    return session_data
import os

def getDataset(fileName):
    session_data = loadFeaturesAndNormalize(fileName)

    allDat = []
    trueSentences = []
    seqElements = []

    for x in range(len(session_data['inputFeatures'])):
        allDat.append(session_data['inputFeatures'][x])
        trueSentences.append(session_data['transcriptions'][x])

        thisTranscription = str(session_data['transcriptions'][x]).strip()
        thisTranscription = re.sub(r'[^a-zA-Z\- \']', '', thisTranscription)
        thisTranscription = thisTranscription.replace('--', '').lower()
        addInterWordSymbol = True

        phonemes = []
        for p in g2p(thisTranscription):
            if addInterWordSymbol and p==' ':
                phonemes.append('SIL')
            p = re.sub(r'[0-9]', '', p)  # Remove stress
            if re.match(r'[A-Z]+', p):  # Only keep phonemes
                phonemes.append(p)

        #add one SIL symbol at the end so there's one at the end of each word
        if addInterWordSymbol:
            phonemes.append('SIL')

        seqLen = len(phonemes)
        maxSeqLen = 500
        seqClassIDs = np.zeros([maxSeqLen]).astype(np.int32)
        seqClassIDs[0:seqLen] = [phoneToId(p) + 1 for p in phonemes]
        seqElements.append(seqClassIDs)
```

```
    newDataset = {}
    newDataset['sentenceDat'] = allDat
    newDataset['transcriptions'] = trueSentences
    newDataset['phonemes'] = seqElements
    newDataset['blockIdx'] = session_data['blockIdx']

    timeSeriesLens = []
    phoneLens = []
    for x in range(len(newDataset['sentenceDat'])):
        print(newDataset['phonemes'][x].shape)
        timeSeriesLens.append(newDataset['sentenceDat'][x].shape[0])
        zeroIdx = np.argwhere(newDataset['phonemes'][x]==0)
        phoneLens.append(zeroIdx[0,0])

    newDataset['timeSeriesLens'] = np.array(timeSeriesLens)
    newDataset['phoneLens'] = np.array(phoneLens)
    newDataset['phonePerTime'] = newDataset['phoneLens'].astype(np.float32) / n
    return newDataset
import nltk
nltk.download('averaged_perceptron_tagger_eng')

trainDatasets = []
testDatasets = []
competitionDatasets = []

dataDir = '/Users/ksrivatsan/SpeechBCI/competitionData'

for dayIdx in range(len(sessionNames)):
    print(dayIdx)
    trainDataset = getDataset(dataDir + '/train/' + sessionNames[dayIdx] + '.ma
    testDataset = getDataset(dataDir + '/test/' + sessionNames[dayIdx] + '.mat'

    trainDatasets.append(trainDataset)
    testDatasets.append(testDataset)

    if os.path.exists(dataDir + '/competitionHoldOut/' + sessionNames[dayIdx] +
        dataset = getDataset(dataDir + '/competitionHoldOut/' + sessionNames[da
        competitionDatasets.append(dataset)
dataDir = '/Users/ksrivatsan/SpeechBCI/competitionData'

competitionDays = []
for dayIdx in range(len(sessionNames)):
    if os.path.exists(dataDir + '/competitionHoldOut/' + sessionNames[dayIdx] +
        competitionDays.append(dayIdx)
print(competitionDays)
```

```
import pickle

allDatasets = {}
allDatasets['train'] = trainDatasets
allDatasets['test'] = testDatasets
allDatasets['competition'] = competitionDatasets

with open('/Users/ksrivatsan/SpeechBCI/data/ptDecoder_ctc', 'wb') as handle:
    pickle.dump(allDatasets, handle)
import matplotlib.pyplot as plt

plt.figure()
#
plt.imshow(trainDatasets[0]['sentenceDat'][10].T, clim=[-1,1]) #10th sentence c
print(trainDatasets[0]['sentenceDat'][10].shape)
plt.show()

plt.figure(figsize=(8,2))
plt.plot(trainDatasets[0]['sentenceDat'][10][:,0]) # all time steps for 1st neu
plt.plot(trainDatasets[0]['sentenceDat'][10][:,128]) # all time steps for 128th
plt.xlim([0,300])
plt.ylim([-1,3])
plt.show()

\item \textbf{Code for GLM fit (python library)}
import matplotlib.pyplot as plt
import numpy as np
import numpy as np
print(np.__version__)
from pyglmnet import GLM, simulate_glm
import pickle

import scipy.io as sio
import scipy.sparse as sps
import warnings

# returns y_train, y_test, each lists of blocks of trials
def get_train_test():
    with open("/Users/ksrivatsan/SpeechBCI/data/ptDecoder_ctc", "rb") as file:
        data = pickle.load(file)

    print(len(data['train']))
    y_train = []

    day_0_train = data['train'][0]
    print(day_0_train.keys())
```

```
    unique_blocks = set(day_0_train['blockIdx'])
    for block in unique_blocks:
        indices = np.where(day_0_train['blockIdx'] == block)[0]
        sentences = [day_0_train['sentenceDat'][j][:, :128] for j in indices]
        y_block = np.concatenate(sentences, axis = 0)
        y_train.append(y_block)

    y_test = []
    day_0_test = data['test'][0]
    unique_blocks = set(day_0_test['blockIdx'])
    for block in unique_blocks:
        indices = np.where(day_0_test['blockIdx'] == block)[0]
        sentences = [day_0_test['sentenceDat'][i][:, :128] for i in indices]
        y_block = np.concatenate(sentences, axis = 0)
        y_test.append(y_block)

    return y_train, y_test

y_train, y_test = get_train_test()
print(len(y_train))
print(len(y_test))
from pyglmnet import GLM, simulate_glm

y_train = y_train[0]
# np.vstack(y_train)
y_test = np.vstack(y_test)
neurons = 128

# need to make X_ train the number of
print(y_train.shape)
R2scores = []
lags = [10, 100, 200]
average_scores = []

for lag in lags:
    print(lag)
    R2scores = []
    for neuron in range(100):
        if(neuron % 10):
            print(neuron)
        y = y_train[:, neuron]
        X = np.vstack(y[(i - lag):i] for i in range(lag, y.shape[0]))
        print(X.shape)
        X = np.concatenate((np.zeros((lag, lag)), X), axis = 0) # pad at the to
        glm = GLM(distr='poisson', score_metric='pseudo_R2', reg_lambda=0.01)
        glm.fit(X, y)
```

```
        score = glm.score(X, y)
        print(score)
        R2scores.append(score)
    average_scores.append(sum(R2scores) / len(R2scores))
print(average_scores)
plt.plot(lags, average_scores)
plt.title("R2 scores from a Poisson Generalized Linear Model for a sample of 10
plt.xlabel("History (K)")
plt.ylabel("R2 score")
```

- **Code for extracting PSID latent states**

```
import os
def saveLatentStates(latents, filename, day=0, time_first=False):
    with open("/Users/ksrivatsan/SpeechBCI/data/ptDecoder_ctc", "rb") as file:
        data = pickle.load(file)

    if time_first is False:
        latents = latents.T

    print(data['train'][day].keys())
    session = {}
    session['train'] = data['train'][day]
    print(session['train'].keys())
    session['train']['sentenceDat_latent'] = []
    i = 0

    for sentence in data['train'][day]['sentenceDat']:
        j = i + sentence.shape[0]
        session['train']['sentenceDat_latent'].append(latents[i:j, :])
        i = j
    for sentence in data['test'][day]['sentenceDat']:
        j = i + sentence.shape[0]
        session['train']['sentenceDat_latent'].append(latents[i:j, :])
        i = j
    print(len(session['train']['sentenceDat_latent']))

    session['train'] = [session['train']]

    with open(os.path.join('/Users/ksrivatsan/SpeechBCI/data', filename), 'wb')
        pickle.dump(session, file)
    return session

x_all = np.concatenate(x_block, axis =0)
session = saveLatentStates(x_all, 'ptDecoderLatentPSID_ctc_32', day=0, time_fi
```

```
\item \textbf{Code for assessing EM goodness of fit}

import matplotlib.pyplot as plt
import math
import numpy as np
import importlib
import os
import scipy.io as sio
import sklearn

delta = 0.02
latent_dim = 16
C = 128

speechBCI_data_dir = '/Users/ksrivatsan/SpeechBCI/competitionData'
EM_data_dir = '/Users/ksrivatsan/EM/multiscaleEM'

split = 'train'
day = 't12.2022.04.28.mat'
## Assess MSF (multiscale filter) Model Goodness-of-fit

##### Let us see if the training data was fit well
data_day = sio.loadmat(os.path.join(speechBCI_data_dir, split, day))
tx = data_day['tx1'].squeeze()
print(tx.shape)

N_day = np.empty((0, 128))
for i in range(tx.shape[0]):
    N_trial = tx[i][:, :128]
    N_day = np.concatenate((N_day, N_trial), 0); # stack trials vertically in t
N_day = N_day.transpose().astype(float)
N_day.shape
#CIF parameters
#### Load results.mat file from EM
Theta = sio.loadmat(os.path.join(EM_data_dir, 'results.mat'))['resultsEM']['The
X_predictions = sio.loadmat(os.path.join(EM_data_dir, 'results.mat'))['resultsE
X_smoothed = sio.loadmat(os.path.join(EM_data_dir, 'results.mat'))['resultsEM']
T = X_predictions.shape[1]

ones = np.expand_dims(np.ones(X_predictions.shape[1]), 0)
x_smoothed_bias = np.concatenate((ones, X_predictions), axis=0)
# now each x is [1; x'] '

Theta_T = Theta.T

Nc = np.zeros((C, T))
```

```
for t in range(T):
    Nc[:, t] = np.exp(np.matmul(Theta_T, x_smoothed_bias[:, t])) * delta
from sklearn.metrics import roc_auc_score
import matplotlib.pyplot as plt

N_day_corrected = (N_day >= 1)

# flattened_predictions = Nc.flatten()   # Shape: (C * T,)
# flattened_N_true = N_day_corrected.flatten()  # Shape: (C * T,)

# fpr, tpr, thresholds = roc_curve(flattened_N_true, flattened_predictions)
# print(thresholds.shape)
# roc_auc = auc(fpr, tpr)

# # Plot the ROC curve
# plt.figure()
# plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f
# plt.plot([0, 1], [0, 1], color='gray', linestyle='--', label='Random Guess')
# plt.xlabel('False Positive Rate')
# plt.ylabel('True Positive Rate')
# plt.title('Receiver Operating Characteristic (ROC) Curve')
# plt.legend(loc='lower right')
# plt.grid()
# plt.show()
def get_ind_mean_firing_rates(N_day, neurons):
    rates = []
    for neuron in neurons:
        rates.append(np.mean(N_day[neuron, :]))
    return sum(rates) / len(rates)

pp =  np.zeros(C)
for c in range(C):
    pp[c] = 2 * roc_auc_score(N_day_corrected[c, :], Nc[c, :]) - 1

# compare where its PP is > 0.2 with overall mean firing rates
print(f"Mean firing rates for PP > 0.4 is {get_ind_mean_firing_rates(N_day, np
print(f"Mean firing rates for all neurons is {get_ind_mean_firing_rates(N_day,

### We can plot mean firing rates as a function of the minimum threshold of PP
thresholds = np.arange(0, 1, 0.05)
firing_rates = np.array([get_ind_mean_firing_rates(N_day_corrected, np.where(pp
plt.plot(thresholds, firing_rates)
plt.xlabel("Minimum threshold PP")
plt.ylabel("Average firing rates across neurons", rotation=0, labelpad=20)
```

```
from sklearn.decomposition import PCA

latent_states = X_smoothed.T # get features by time
print(latent_states.shape)

# Apply PCA with 3 components
pca = PCA(n_components=16)
latent_states_pca = pca.fit_transform(latent_states)

print(f"Explained variance with 3 dimensions: {np.sum(pca.explained_variance_ra

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(latent_states_pca[:, 0], latent_states_pca[:, 1], latent_states_pca
ax.set_xlabel('Principal Component 1')
ax.set_ylabel('Principal Component 2')
ax.set_zlabel('Principal Component 3')
ax.set_title('Latent State Dynamics (PCA in 3D)')
plt.show()
```

- **Code for evaluating CV model predictions**

```
import glob
import pickle

files = glob.glob('../logs/speech_logs/speechBaseline4/trainingStats[0-9]*')
cer_rates = []
for file in files:
    with open(file, 'rb') as file:
        data = pickle.load(file)
        cer_rates.append(data["testCER"][-1])

print(sum(cer_rates) / len(cer_rates))

\item \textbf{Code for Feed-forward neural network}

import torch
from torch import nn

class MLPDecoder(nn.Module):
    def __init__(
        self,
        neural_dim, # latent feature dimensionality
        n_classes,
        hidden_dim, # 2 layers?
        dropout = 0,
```

```
        device="cuda",
        strideLen = 4,
        kernelLen = 32,
    ):
        super(MLPDecoder, self).__init__()

        self.neural_dim = neural_dim
        self.n_classes = n_classes
        self.hidden_dim = hidden_dim
        self.dropout = dropout
        self.device = device
        self.strideLen = strideLen
        self.kernelLen = kernelLen

        self.unfolder = torch.nn.Unfold(
            (self.kernelLen, 1), dilation=1, padding=0, stride=self.strideLen
        )

        self.fc1 = nn.Linear(self.kernelLen * neural_dim, hidden_dim) # some co
        self.relu1 = nn.ReLU()

        self.fc2 = nn.Linear(hidden_dim, hidden_dim) # hidden layers
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_dim, n_classes + 1) # phoneme decoding laye

    def forward(self, x, dayIdx):

        stridedInputs = torch.permute(
            self.unfolder(
                torch.unsqueeze(torch.permute(x, (0, 2, 1)), 3)
            ),
            (0, 2, 1),
        )
        embedding = self.fc1(stridedInputs)
        embedding = self.relu1(embedding)

        embedding2 = self.fc2(embedding)
        embedding2 = self.relu2(embedding2)

        dec_seq = self.fc3(embedding2)

        # dec_seq_prob = torch.softmax(dec_seq, 2)
        return dec_seq
```

References

[1] F. Willett, E. Kunz, C. Fan, D. Avansino, G. Wilson, E. Choi, F. Kamdar, M. Glasser, L. Hochberg, S. Druckmann, K. Shenoy, and J. Henderson, "A high-performance speech neuroprosthesis," *Nature*, vol. 620, no. 7976, pp. 1031–1036, Aug. 2023, publisher Copyright: © 2023, The Author(s).

[2] H. Abbaspourazad, M. Choudhury, Y. T. Wong, B. Pesaran, and M. M. Shanechi, "Multiscale low-dimensional motor cortical state dynamics predict naturalistic reach-and-grasp behavior," *Nature communications*, vol. 12, no. 1, pp. 607–607, 2021.

[3] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 369–376. [Online]. Available: https://doi.org/10.1145/1143844.1143891