CS230

A Recurrent Neural Net for Neurons: Continuous Decoding of Intracortical Brain Signals for BMI Applications

Jonathan Zwiebel
Computer Science, Class of 2021
Stanford University
jzwiebel@stanford.edu

Samuel Lurye
Electrical Engineering, Class of 2020
Stanford University
slurye@stanford.edu

Robert Ross
Computer Science, Class of 2021
Stanford University
rross@stanford.edu

Abstract

Our project seeks to create a model for making continuous predictions of arm position based on neuron activity. Our data set consisted of labelled neuron-position sequences measured via surgical implants on monkeys provided by Professor Krishna Shenoy of the Neural Prosthetics Systems Lab. Our works focuses on many-to-many LSTM networks and includes experimentation with different dataset parameters and model targets. We tuned models to predict relative position, instantaneous velocity, and binned velocity and found that the models with relative position as their target gave the best path predictions. We also experimented with different data sampling methods and found that subsequences of 400 - 500 ms provided adequate training speed without compromising the performance of the model. Our final model was able to outperform our baseline Wiener model with a mean squared error on position of 17.5 cm, compared to the Wiener model 32.3 cm.

1 Introduction

Brain machine interfaces (BMIs) have recently emerged from academic labs as viable clinical options for paralyzed patients. Using just thought, patients are able to move robotic arms and computer cursors with increasing precision and dexterity. These neural prosthesis have enormous potential as medical devices for patients with neural degeneration.

The purpose of this project is to develop a deep learning decoder for BMIs. Expanding on existing linear models, this project systematically builds a many-to-many LSTM network that can continuously decode a high-dimensional neural signal into a set of X-Y coordinates. Beginning with a simple RNN, we iteratively derive a tailored architecture and set of hyper-parameters that minimizes prediction error and minimizes training time.

2 Dataset

The data for this project consisted of 11,136 trials [1] of a non-human primate performing radial reach tasks to one of 48 targets.

Alongside the X-Y position of the monkey's hand at every millisecond, the trial included a size 192 binary vector for each time step. This column vector corresponded to the 192 electrode array implanted in the monkey's motor cortex. A 1 in the ith row of the vector indicates that a 'spike' had occured in the neuron adjacent to the ith electrode at a given point in time. Horizontally concatenating these vectors produced a matrix of size 192 x the length of the trial in milliseconds, which varied.

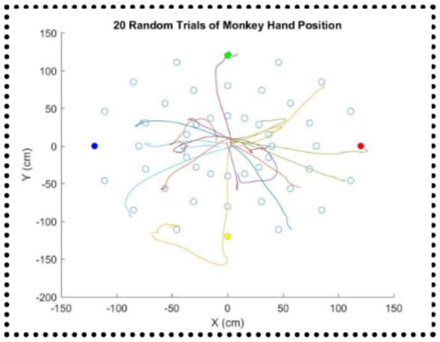


Figure 1- Monkey's hand position in 20 random trials. All 48 targets overlayed.

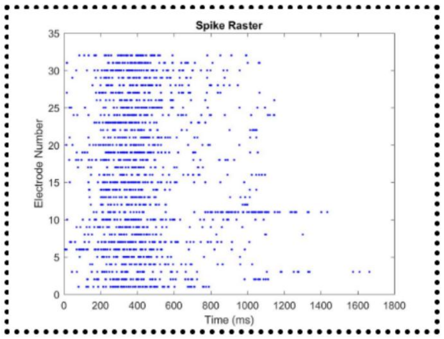


Figure 2- Example neural data. Every blue dot represents a spike at an electrode at a given time point

In order to confirm that the neural data had sufficient variability among different reach directions for our model to differentiate, we performed PCA for a better visualization. We projected the data for 4 orthogonal reach directions onto the 3 top Principal Components and plotted their boundaries. The colors of surfaces correspond to the colored targets in Figure 1.

As a final pre-processing step, we extracted velocity and target vectors from the positional data. The velocity was calculated using a first-order euclidean approximation with a step size of 25ms. The target vectors were derived by normalizing the distance from the X-Y position of the hand to the intended target at any one time. These derived features were used to implement different loss functions during the architecture design phase.

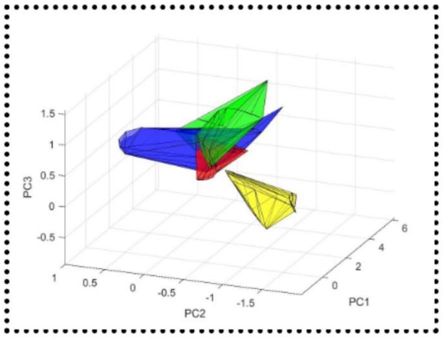


Figure 3- Boundaries of neural data from all 192 electrodes across all trials plotted on top 3 Principal Components

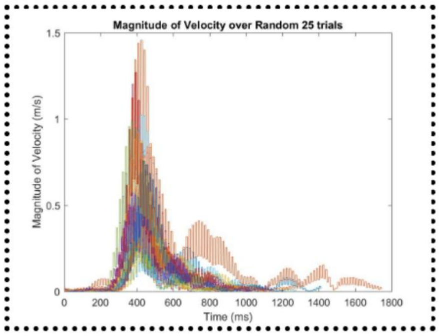


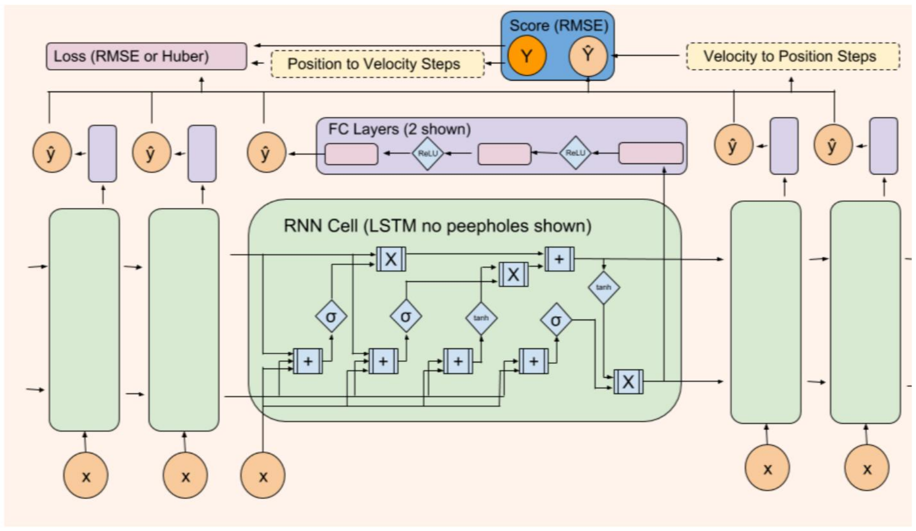
Figure 4- Magnitude of velocity over 25 random trials. Note that increased velocity correlates with increase in neural activity per Figure 2

Given the moderate size of our dataset and the sparsity of our dataset, we opted for a 80-10-10 split between our training, dev, and test sets. We did not use a train-dev set despite the fact that our training was done over shortened sequence and our dev/test analysis was done over sequences of length 800. Given additional time we would have generated a train-dev set.

3 Model Architecture

3.1 Overview

Given the sequential nature of both the input and output data we opted for a many-to-many architecture with a cell for each labelled time interval (1 ms) and unique predictions of position or velocity from each cell. We initially experimented with both plain RNN and GRU cells but found that LSTM cells (shown in green) significantly outperformed them at a marginal cost to training time. We fed the output of each LSTM cell into a series of fully connected ReLU layers (shown in purple) which gradually reduced the size from the cell hidden state size to two. The final fully connected layer did not include any activation function as our output values were real-valued numbers without an explicit bound.


Score (RMSE)
Y
Ŷ
Loss (RMSE or Huber)
Position to Velocity Steps
Velocity to Position Steps
FC Layers (2 shown)
ReLU
ReLU
ŷ
RNN Cell (LSTM no peepholes shown)
σ
tanh
x

3.2 Prediction Targets

Our dataset provided absolute measurements of position measured every 12 ms. Given that our models were trained with sequences starting at arbitrary points in each training sequence and that linear methods had shown prediction of velocity to be successful, we opted to not use the raw absolute position as our target value. Instead we experimented with three different target values for the model to predict: relative position, instantaneous velocity, and binned velocity.

Relative Position: We calculated position values for each timestep relative to the position value at the first measurement in the sequence. Given that our raw dataset included true position we directly fed our prediction and ground truth into our loss function.

Instantaneous Velocity: We calculated measurements of instantaneous velocity by comparing raw position measurements taken at maximum resolution (12 ms). We then repeated each value for 12 outputs so that our values in our input and output sequences both represented measurements taken at 1 ms intervals. We fed these stepped velocity measurements and our prediction into our loss function.

Binned Velocity: This approach resulted in sequences were each timestep represented data over 25 ms blocks. For our input sequences we took the total count of how many times each neuron fired over the 25 ms block to create a size 192 vector of integers. Our output sequences were created using the same approach as instantaneous velocity except they were taken over 25 ms intervals and their values were not repeated. This resulted in input and output sequences 25 times shorter than in the models that used relative position and instantaneous velocity. We fed this binned calculation of velocity and our prediction into our loss function. Our reasoning behind the binned approach was that it would reduce the sparsity of the input data and ensure that neural impulses at similar time steps were counted together.

3.3 Sequence Selection

A major decision point in our models was how to sample our data sequences. Longer sequences result in slower training but better paths when the models are applied to standardized sequence lengths. Additionally it was important that our sequences contained periods of time with motion. Both sequence length (ms) and sequence starting point (ms from start) were hyperparameters in our model.

3.4 Loss Function

We concatenated the predictions made at each timestep to create a sequence of predictions which we fed into our loss functions. We experimented with both MSE loss and Huber loss, a robust loss function similar to MSE loss, in our models. We quickly found that Huber loss resulted in quicker reduction of our loss value, but ultimately did not result in better predictions as scored by our scoring function (3.5), so we used MSE loss for the majority of our training.

3.5 Score Function

In order to standardized the comparison of our models given their different sequence lengths and targets we developed a standard scoring function. Each model, regardless of the length training sequence, was tested on input sequences of 800 ms. The models that outputted velocity had their output stepped forward into position, and the models which used binning had their outputs repeated such that there was a single prediction for each ms of data. In this way, each model provided a predicted set of 800 positions, spaced 1 ms apart. We used simple mean squared error between the raw output values and our prediction values to score each path, and took the average over all training examples as the final model score.

3.6 Clipping

In order to deal with issues of exploding gradients we implemented gradient clipping across all weights in the LSTM cell.

3.7 Rejected Architectures

Bidirectional RNN: While a bidirectional sequence model would have improved predictions on our labeled dataset, it would not be appropriate for continuous decoding as the problem requires that predictions are generated in real-time.

Repeatedly Sampled Deep Networks: We considered constructing training examples of short sequences (50 ms) that used the full 192 x 50 array as an input and a single size 2 vector representing the difference in position as an output. We opted to not use this method as we did not have a good intuitive understanding of how long it would take a neuron firing to propagate into movement. Additionally we were unsure if there was a consistent time period between neural impulse and motor movement across different neurons.

4 Tuning

While optimizing models for each of our three target values we tuned the following hyperparameters:

1. Hidden state size in LSTM cells
2. Learning rate
3. Batch size
4. Initial skip length and sequence length
5. Number of fully connected layers and layer size

The following hyperparameters were left as their default values:

1. Gradient clipping boundaries: -1 to 1

2. LSTM cell activation function: tanh
3. FC layers activation function: ReLU
4. Adam beta 1: 0.9
5. Adam beta 2: 0.999
6. Adam ϵ: 10^-8

4.1 Hidden State Size

We initially started training with 200-300 hidden state size in the LSTM cells. The loss experienced frequent jumps after 500 epochs and the predicted coordinates were poor even after long training runs. We lowered the hidden state size to 50 to achieve faster training and found that the loss decreased mostly monotonically and predictions improved. We anticipate that the size 200-300 network would eventually achieve superior performance but at the expense of a very long training process (2500 epochs). Cells with a hidden state of size 50 produced functional performance after only 1000 epochs. **Final Parameter: 50**

4.2 Learning Rate

Given the natural variability in loss decrease while using the Adam Optimizer, we had to compensate by using a smaller learning rate. We found that anything greater than the default 0.001 would cause frequent jumps in the loss **Final Parameter: 0.0005**

4.3 Batch Size

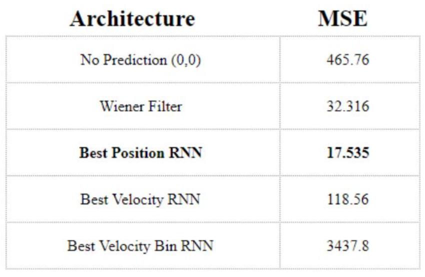
Given the sparsity of our input dataset we opted for large batch sizes. We found that batch sizes of 4096 or greater resulted in large jumps in training loss. We used the largest batch size possible that gave a mostly monotonically decreasing loss. **Final Parameter: 512**

4.4 Number of FC Layers and Layer Size

We experimented first with a large number of FC layers but found that only a single hidden layer (along with the output layer) was sufficient to produce good results. We additionally found that without dropout, more than 1 hidden layer resulted in severe overfitting of our data. Given that the size of the hidden state played a large part in our FC layer size, we did not have much time to experiment with different values. We experimented both with values near the hidden state size (50) and values near the output layer size (2) and found that 30 was the best layer size. **Final Parameter: 1 hidden FC layer with 30 units**

5 Results

The Many-To-Many RNN with positional loss was the top performing deep learning model. Trained with the optimal hyper-parameters outlined above, the model predicted paths better than our baseline linear model, the Wiener Filter. We scored our models using Root Mean Squared Error between predicted pathways and actual monkey hand movements.

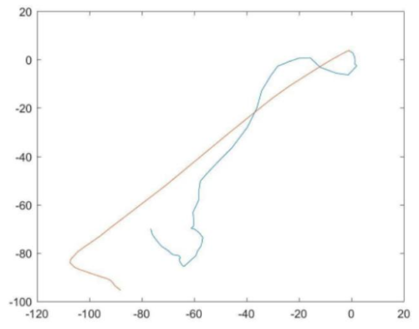


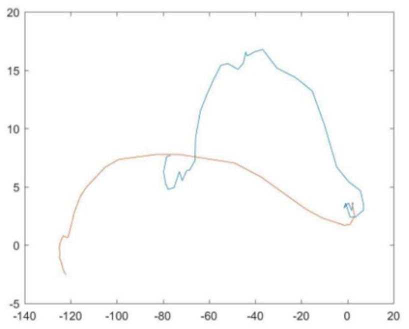
Architecture	MSE
No Prediction (0,0)	465.76
Wiener Filter	32.316
Best Position RNN	**17.535**
Best Velocity RNN	118.56
Best Velocity Bin RNN	3437.8

Figure 5- Table of RMSE score values for each model

These score values were reflected in the subjective appearance of the paths predicted by each model. Below, we have included some average performances from our model and the baseline Wiener Filter.

Wiener Filter Predictions:





Best LSTM Model Predictions:

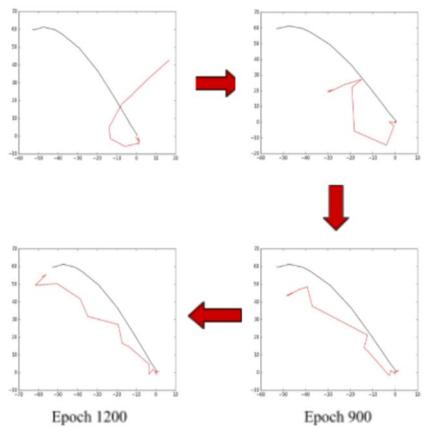


Figure 6a - Sample prediction over 1200 epochs using RNN (red) vs true position (black)

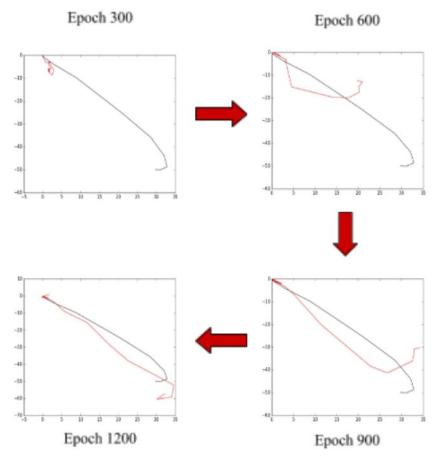


Figure 6b - Sample prediction over 1200 epochs using RNN (red) vs true position (black)

6 Conclusion

We found that position was the best target value to optimize for, and that models that optimized on velocity suffered from overfitting on the training set or simply did not train well. Our best LSTM model was able to outperform the naive Wiener filter and produced usable predictions of position. We did find that the predicted paths were excessively rough to serve as direct values for a cursor, so additional post-processing of their values would be needed before they could be used for cursor control.

7 Future Work

Given additional training time we would use a more sophisticated loss and score function. In particular we would like to penalize smoothness of the prediction and develop a loss function that depends on both position and velocity. Additionally we would like to experiment with RNN architectures that explicitly pass predicted outputs as inputs into future cells (ex: echo networks).

One other method we considered but did not attempt was to develop a dimensionality reduction of the input data to deal with the issue of artificial sparsity in the input data. One method we considered was using an attention model to develop an 'encoding' of neural firing states similar to the methods used for NLP. The logic behind this method is that there are many related neural states or states that follow each other. Another method was simply to pass PCA reduced vectors in as input.

Finally, given additional resources we would attempt to take measurements of subjects performing motion tasks with more complex motion than the radial ones used for our dataset.

8 Code and Dataset

Our code can be found at www.github.com/JonathanZwiebel/cs230-project. Our dataset is not currently available to the public.

References

[1] Sussillo, D., Nuyujukian, P., Fan, J. M., Kao, J. C., Stavisky, S. D., Ryu, S., & Shenoy, K. (2012). A recurrent neural network for closed-loop intracortical brain–machine interface decoders. Journal of Neural Engineering, 9(2), 026027. http://doi.org/10.1088/1741-2560/9/2/026027

[2] Barak, O. (2017). Recurrent neural networks as versatile tools of neuroscience research. Current Opinion in Neurobiology, 46, 1-6. doi:10.1016/j.conb.2017.06.003

[3] Güçlü, U., & van Gerven, M. A. J. (2017). Modeling the Dynamics of Human Brain Activity with Recurrent Neural Networks. Frontiers in Computational Neuroscience, 11, 7. http://doi.org/10.3389/fncom.2017.00007

[4] Kao, J., Stavisky, S., Sussillo, D., Nuyujukian, P., & Shenoy, K. (2014). Information Systems Opportunities in Brain–Machine Interface Decoders. IEEE.

[5] Olah, C. (2015, August 27). Understanding LSTM Networks. Retrieved from colah.github.io/posts/2015-08-Understanding-LSTMs/