Neuronal Voltage Activity Detection From Line Mappings

Deep Learning Project - Final Report
Chinmayan Pradeep (cpk286), Divyansh Agarwal (da3245), Rohaan Advani(rna3535)

Link to the Code Repository - DL Final Project

Introduction

The brain's ability to process information is heavily dependent on the intricate dynamics of neural activity. Neurons, the basic units of the nervous system, communicate through electrical signals known as action potentials or spikes. These signals originate in various parts of the neuron, including the **cell body (soma), axons, and dendrites**, tree-shaped structures that receive input from other neurons. Dendrites play a crucial role in the integration of synaptic signals and their activity can provide critical information on neural function. This project focuses on detecting *neuronal events* (spikes) from time-series voltage data recorded along dendritic regions.

To observe neuronal activity, advanced imaging techniques such as **voltage imaging** are used. Voltage imaging enable measurement of changes in membrane potential in different regions of the neuron. When applied to dendrites, voltage imaging generates high-resolution **spatial and temporal data** in the form of line mappings. Line mappings are essentially voltage traces captured along specific dendritic lines, providing a continuous record of electrical fluctuations over time.

Detecting events in these traces is inherently challenging due to noise, variability in spike characteristics (e.g., shape and amplitude), and overlapping background signals. Traditional signal processing techniques often fail to generalize under diverse conditions, while manual annotation remains laborious and impractical for large datasets.

To overcome these challenges, deep learning offers a powerful solution. Deep learning models can effectively capture temporal patterns and subtle features that signify spikes. They are suited for this task as they excel at learning complex signal dynamics and distinguishing noise from meaningful events.

This project aims to automate neuronal event detection using deep learning techniques on time series voltage data and compare it against the performance of a classical machine learning technique. By providing a scalable and accurate spike detection method, this work has significant implications for analyzing neural activity, advancing our understanding of synaptic integration, and facilitating research in **neuroinformatics**.

Related Work

This section reviews significant research in event detection from time series data, particularly focusing on deep learning approaches and their applications to neural event detection. Models such as RNNs, LSTMs, GRUs, TCNs, and Transformers, along with spectrogram-based techniques, have been explored to address challenges posed by noisy temporal data.

Event Detection in Neural Activity Data

Deep learning has emerged as a transformative tool for detecting spikes in neural activity data. For instance, van der Molen et al. (van der Molen et al. 2023) introduced RT-Sort, an algorithm that utilizes a convolutional neural network (CNN) for real-time action potential detection and sorting, achieving millisecond latencies. This approach underscores the potential of CNNs in processing neural voltage traces, facilitating scalable and robust automated spike detection models.

Spectrogram-based Analysis for Time Series

Spectrogram-based methods transform temporal signals into time-frequency representations, enabling deep learning models to capture complex patterns. Choi et al. (Choi, Fazekas, and Sandler 2016) successfully applied CNNs to spectrograms for event detection in audio signals. Similarly, Nadalin et al. (Nadalin et al. 2021) developed a convolutional neural network trained on spectrogram images to detect spike ripple events in EEG data, demonstrating the adaptability of spectrogram-based analysis for physiological signals. By leveraging the spectral properties of time series data, these techniques provide a reliable framework for identifying distinct temporal events.

Recurrent Neural Networks and Variants

Recurrent Neural Networks (RNNs) and their variants, such as long-short-term memory (LSTM) networks and Gated Recurrent Units (GRUs), have been widely used for the detection of time series events. Malhotra et al. (Malhotra et al. 2015) demonstrated the effectiveness of LSTMs in learning long-term dependencies for anomaly detection in time series data. GRUs, introduced by Chung et al. (Chung et al. 2014), offer similar performance with fewer parameters, making

them computationally efficient for tasks involving noisy and irregular signals. These architectures excel at capturing temporal dependencies critical for spike detection in voltage traces.

Temporal Convolutional Networks (TCNs)

Temporal Convolutional Networks (TCNs) (Bai, Kolter, and Koltun 2018) have gained prominence as an alternative to RNNs for sequence modeling. TCNs use dilated causal convolutions to capture long-range temporal dependencies while maintaining computational efficiency. Bai et al. (Bai, Kolter, and Koltun 2018) demonstrated that TCNs outperform LSTMs and RNNs in sequence tasks such as speech and video processing. Their ability to combine local pattern recognition with a large receptive field makes TCNs suitable for event detection in neural voltage signals.

Transformers and Attention Mechanisms

The introduction of Transformers (Vaswani et al. 2017) revolutionized time series modeling by enabling global context learning through self-attention mechanisms. Li et al. (Lim et al. 2021) proposed the Time Series Transformer (TST), which achieved state-of-the-art results in forecasting and anomaly detection by focusing on salient features in the signal. These models have shown remarkable performance in identifying sparse events, such as spikes, by efficiently capturing global temporal dependencies.

Hybrid and Traditional Approaches

Hybrid models combining deep learning with traditional signal processing techniques have also been explored for event detection. Pachitariu et al. (Pachitariu, Steinmetz, and Kadir 2016) introduced Kilosort, a method that uses PCA and clustering to extract spike templates from voltage recordings. Sparse coding techniques have also been employed to represent neural signals with minimal basis functions, effectively reducing noise while preserving critical event features.

Dataset

The data were generated by the *Losonczy Lab* at Columbia University as part of their ongoing research. We have about 10 experiments conducted with each experiment having about 15 scans, which comes up to 159 scans in total. Each scan has two types of values that we focus on, *denoised trace* and *spike interval*, with each scan being around 100,000 long data points. The denoised trace contains information about voltage data captured by the high speed microscope and is denoised by a separate model. The spike interval has binary classification for each denoised trace point, 0 for not an event and 1 indicating event. The data is highly imbalanced with majority of data being not an event. The spike interval data has been manually annotated by the Losonczy lab.

The raw data contain about 500 scans from many more experiments, but it is all noisy data and needs denoising with some better models. Due to this noise, this raw stream cannot be used with current model. A future proposal is to use the current models and test out the performance on noisy data and see the models ability to handle the complex data patterns.

Mel Spectrogram

This problem clearly requires a technique that can capture some neighboring information. This is the motivation for using models like Transformers, LSTMs, etc. However, what if we wanted to use a simpler model like an SVM? The goal then is to somehow capture information about neighboring interaction via data augmentation. This is the motivation behind using a Mel Spectrogram (Figure 1) — it translates signals into a visual representation of its distribution across different frequencies over time. According to the lab, the sampling rate of the signal was 3,328 samples per second. The signal is divided into overlapping frames 12 samples each, with a hop_length of 1, ensuring maximum temporal resolution by shifting only one sample between frames. We manually tweaked the temporal and frequency resolution to ensure that the events are appropriately represented in the spectrograms. Each frame undergoes a Fourier Transform to extract frequency components, which are grouped into 7 mel bands using triangular filters.

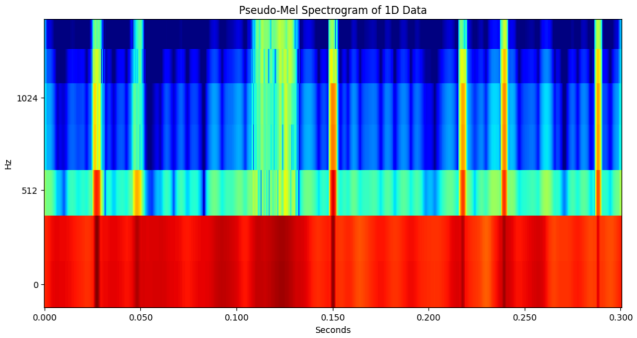

Figure 1: A sample spectrogram window. The event locations can be qualitatively ascertained.

After conversion to Mel Spectrograms, we can either save the images as .jpgs or save the individual frequency values (seen as colors in the image). The latter essentially creates seven features per sample. We also explored how separable these frequency bins values are using Principal Component Analysis (as that would be a direct indicator of how effective the SVM can be). Figure 2 shows a sample Principal Component Analysis graph for a small subset of data. Since SVMs are limited to CPU execution, we always worked with a subset of data totaling about 100K samples comprising of parts from every scan.

Data Sampling

For models that intrinsically capture inter-sample interaction either sequentially, through attention or via convolution, we opted to directly use the denoised trace value instead of the generated Mel Spectrogram frequency bins. Each scan from the labeled data is split into three slices, for train, validation and test dataset. First, 0 to 70% is training data, 70% to 85% is validation set, 85% to 100% is test set. All the scans

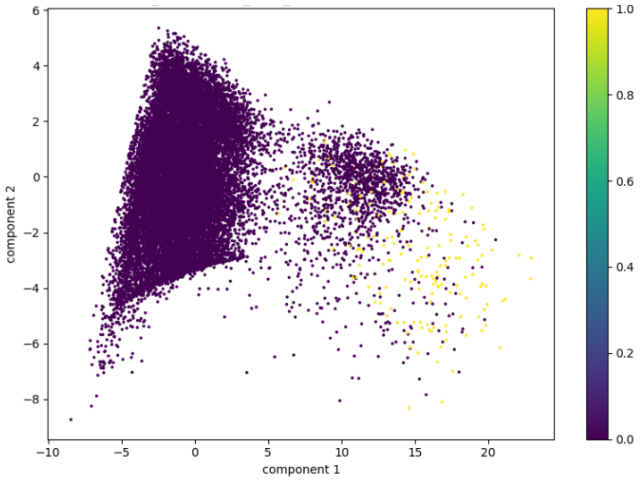

Figure 2: PCA of a small sample of the total data, maintaining original distribution of 1s and 0s.

are passed through and added to training, validation and test sets. The data in each set is then windowed, the long traces are split into windows of 500 data points, with 50 point overlap.

Methods

This project primarily explored applying Support Vector Machine, Temporal Convolution Networks, SequentialRNN-LSTM and Transformer architectures to the data. We also experimented with applying CNNs, VGG and MobileNet architectures on mel spectrogram images, and MLP and xgBoost on mel spectrogram frequency bins. These are omitted in this report due to low to moderate performance and to limit the scope.

Support Vector Machine (SVM)

Model Description Before exploring deep learning techniques, a Support Vector Machine (SVM) was employed as a baseline machine learning model for classification. SVMs are well-suited for binary classification tasks, particularly when the dataset may have overlapping classes or require decision boundaries that are not strictly linear. An SVM with a Radial Basis Function (RBF) kernel was selected due to its ability to map data to a higher-dimensional feature space, improving the model's capacity to separate non-linearly separable data.

Loss Function The SVM utilized a hinge loss function implicitly through the objective of maximizing the margin between support vectors and the decision boundary. The model was configured with balanced class weights to account for any class imbalance in the dataset.

Hyperparameters

- **Regularization Parameter (C):** A grid search was conducted over values $[0.1, 1, 10, 100]$ to find the optimal trade-off between maximizing the margin and minimizing classification error. The final value of $C = 100$ achieved the best performance.
- **Kernel:** The RBF kernel was chosen for its ability to handle non-linear decision boundaries by projecting the data into a higher-dimensional space.
- **Gamma:** The parameter `scale` was selected to standardize the kernel's behavior by normalizing feature values, ensuring the model appropriately captures patterns without overfitting.
- **Class Weight:** The `balanced` parameter was used to mitigate class imbalance by assigning weights inversely proportional to class frequencies.
- **Cross-Validation ($k = 5$):** A 5-fold cross-validation ensured the robustness of hyperparameter tuning and reduced the risk of overfitting.

Training The SVM model underwent an extensive grid search over the parameter space to identify the optimal combination of hyperparameters with F1 score as the optimization criterion.

The final model was retrained on the 100K samples from the dataset using the best hyperparameters ($C = 100$, `kernel=rbf`, and `gamma=scale`). The optimized SVM achieved competitive results on the test set, with performance evaluated using a detailed classification report that included precision, recall, and F1 score metrics.

Temporal Convolution Network (TCN)

Model Description The Temporal Convolution Network (TCN) was selected for its ability to model sequential dependencies through causal convolutions and dilations, which enable capturing long-term temporal relationships without recurrent connections. Unlike RNN-based architectures, TCN processes sequences in parallel and maintains a constant path length for all timesteps, resulting in faster training and inference.

Loss Function Categorical cross-entropy loss (`nn.CrossEntropyLoss`)

Hyperparameters

- **Input Size (1):** The input represents univariate time-series data, so a single channel is sufficient.
- **Number of Channels ([32, 64, 128, 256]):** Four layers with increasing channels allow the model to learn hierarchical temporal features with growing complexity.
- **Kernel Size (3):** A small kernel size was chosen to balance computational efficiency and the ability to capture local temporal patterns.
- **Dropout Rate (0.002):** A minimal dropout rate was used to prevent overfitting while retaining most temporal information.
- **Learning Rate (0.001):** Optimized for the Adam optimizer to ensure stable convergence.
- **Weight Decay (1×10^{-4}):** Used to regularize the model and reduce overfitting.
- **Batch Size (32):** A moderate batch size ensures computational efficiency and robust gradient estimates.

Training The TCN model was trained for a maximum of 50 epochs with early stopping, where training ceased if the validation loss did not improve for 10 consecutive epochs. The Adam optimizer was employed with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, coupled with weight decay to regularize the model and mitigate overfitting.

The best model was saved based on the lowest validation loss. The TCN achieved competitive performance, leveraging its ability to capture both local and long-range dependencies in time-series data.

Transformer

Model Description The Transformer model was chosen for its ability to model long-range dependencies using self-attention mechanisms. Unlike recurrent models, which process data sequentially, Transformers process entire sequences in parallel, making them computationally efficient for longer sequences. The self-attention mechanism allows the model to weigh the importance of all timesteps, making it particularly suited for complex temporal relationships.

The architecture comprises three Transformer encoder layers, each with multi-head self-attention and feedforward sub-layers. Positional encodings were added to the input embeddings to preserve sequence order, ensuring the model retains positional information.

Loss Function The same categorical cross-entropy loss function (`nn.CrossEntropyLoss`) was applied, averaged across all time steps.

Hyperparameters

- **Model Dimension (d_{model}, 64):** The dimension of the input embeddings was set to balance expressiveness and computational efficiency.
- **Number of Encoder Layers (3):** Three layers were chosen to provide sufficient modeling capacity without overfitting.
- **Number of Attention Heads (4):** Four heads allow the model to focus on multiple relationships in the data simultaneously.
- **Feedforward Dimension (128):** The feedforward layers were expanded to ensure the model has sufficient capacity to learn non-linear relationships.
- **Dropout Rate (0.1):** A slightly higher dropout rate than the RNN was chosen to counteract the higher capacity of the Transformer.
- **Learning Rate (0.001):** Optimized for the Adam optimizer to ensure stable and efficient training.
- **Batch Size (32):** Maintained for consistency across models and to balance training efficiency.

Training The Transformer was trained using the Adam optimizer, with the same default parameters as the SequentialRNN-LSTM. Similar to the RNN model, early stopping with a patience of 20 epochs was employed. However, a more complex training dynamic could be explored by incorporating learning rate schedulers such as the *ReduceLROnPlateau* scheduler for better convergence.

The training process consisted of:

1. **Input Preparation:** Time-series data were embedded into a higher-dimensional space using a learnable embedding layer. Positional encodings were added to this embedding to account for sequence order.
2. **Forward Pass:** The data were passed through three Transformer encoder layers, generating sequence-level representations. These representations were processed by a linear classification layer to produce logits.
3. **Loss Computation and Optimization:** Similar to the RNN model, loss was computed and backpropagated, and weights were updated via the Adam optimizer.
4. **Validation:** Validation loss was monitored, and the model state was saved upon improvement.
5. **Grid Search:** This helped select the best hyperparameters required for getting the highest F1 score by exploring all combinations.

d_model	nhead	num_encoder_layers	F1 Score
64	4	3	0.8488
64	4	6	0.8660
64	8	3	0.8428
64	8	6	0.8611
128	4	3	0.8401
128	4	6	0.8529
128	8	3	0.8494
128	8	6	0.8602

Table 1: Grid Search for Transformer Hyperparameters

SequentialRNN-LSTM

Model Description The SequentialRNN-LSTM model employs a bidirectional Long Short-Term Memory (LSTM) network for sequence labeling. LSTMs are particularly effective for this task because they are designed to capture temporal dependencies and are well-suited for handling sequential data with variable lengths. The bidirectional nature enhances the model's ability to learn both past and future dependencies, improving the context captured for each timestep.

The architecture consists of a single bidirectional LSTM layer followed by a fully connected linear layer. The LSTM outputs hidden states for each timestep, which are concatenated (due to the bidirectional setup) and fed into the linear layer for timestep-level classification.

Loss Function categorical cross-entropy loss function (`nn.CrossEntropyLoss`)

Hyperparameters

- **Input Size (1):** The input consists of univariate time-series data, so a single input channel suffices.
- **Hidden Size (64):** Chosen as a balance between model capacity and computational cost; it allows sufficient learning of temporal features without overfitting.
- **Number of Layers (2):** A single LSTM layer with two directions ensures adequate modeling of temporal dependencies without making the network overly deep.

- **Output Size (2):** Set to the number of classes in the binary classification task.
- **Batch Size (32):** A moderate batch size balances memory efficiency and convergence speed.

Training The model was trained for a maximum of 50 epochs using the Adam optimizer (`optim.Adam`). Early stopping was employed, halting training when validation loss did not improve for consecutive epochs. During training, the learning rate remained constant at 0.001, and no learning rate scheduler was applied.

The table for the grid search was too big to add to this report you can find it at this link

Results

Support Vector Machine (SVM)

Multiple SVMs were constructed by varying the number of samples, undersampling the majority class to achieve different balances in the dataset classes, and adding a window size (7 - 35 features per sample). The best F1 score we could achieve via threshold was around **0.67** as seen in Figure 3. This was the number to beat when moving on to our deep learning methods.

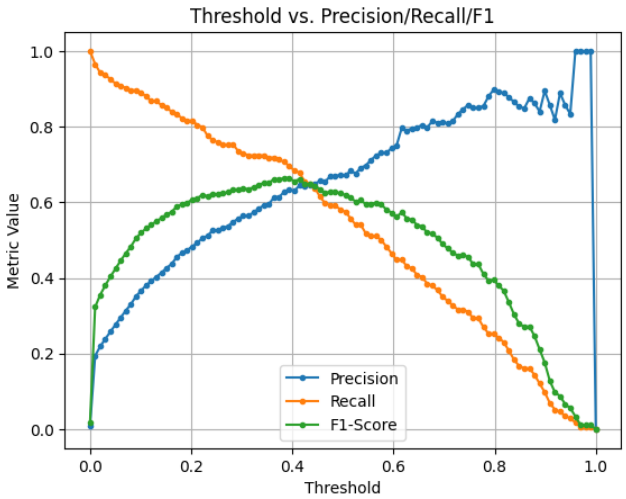

Figure 3: A plot of Precision, Recall and F1 Score at various threshold values. This model was trained with a window size of 3 (21 features per sample).

Temporal Convolution Network (TCN)

Despite exploring various hyperparameter configurations and architectural adjustments, the Temporal Convolutional Network (TCN) achieved a modest F1-score of 0.82. This suggests that the TCN, in its current configuration, may not be well-suited for capturing the underlying patterns in the data, or that the available dataset size is insufficient for this model's complexity. Further investigation with larger datasets or alternative architectures might be necessary to achieve improved performance.

Transformer

As per table 1, we can conclude that the combination of hyperparameters that worked best for this dataset were Model Dimensions = 64, Number of Attention Heads = 4, and Number of Encoder Layers = 6. The F1 score for this combination of hyperparameters came out to be 0.866.

SequentialRNN-LSTM

The Long Short-Term Memory (LSTM) model demonstrated robust performance across diverse datasets, exhibiting adaptability to varying window sizes. To optimize the model's hyperparameters, we conducted an exhaustive grid search, systematically varying the learning rate, dropout rate, and hidden layer size to identify the combination yielding the best performance.

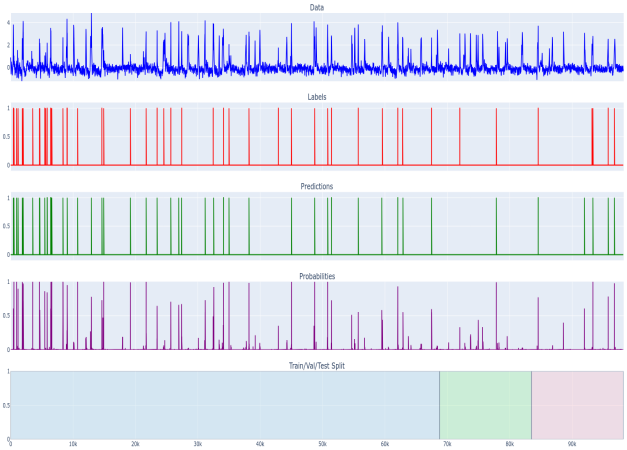

Figure 4: Stacked Plots Denoised Trace, Ground Truth, Prediction, Probabilities and Dataset Split. To view interactive version, click on the graph, and download the .html file.

We also built an interactive stack of all information using plotly with the following elements — denoised trace, event labels, predicted probabilities, predicted mask after thresholding and a train-val-test mask. For a healthy model, we expect the following behavior:

- Near 100% accuracy in the training set.
- Generalizing well to unseen test set.

As seen in the interactive plot, the model displays healthy behavior and does not seem to be overfitting or underfitting. This claim is further backed by Figure 5 — we systematically varied the training set size from 10% to 100% in increments of 10% and evaluated the model's performance using the F1 score. As illustrated in the figure, the results demonstrate that the model learns effectively from the dataset and exhibits no signs of over-fitting.

Conclusion

In this project, we explored and evaluated several machine learning models for time-series classification, includ-

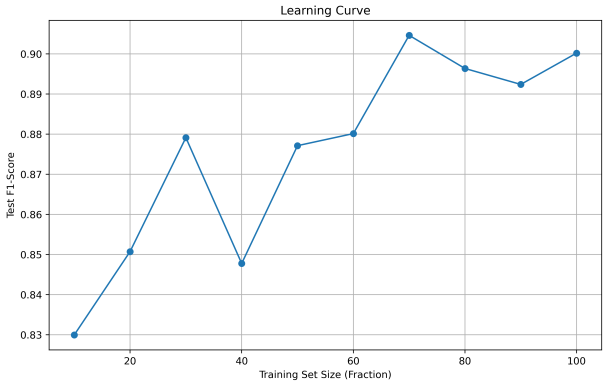

Figure 5: Test Performance against Varying Training Size.

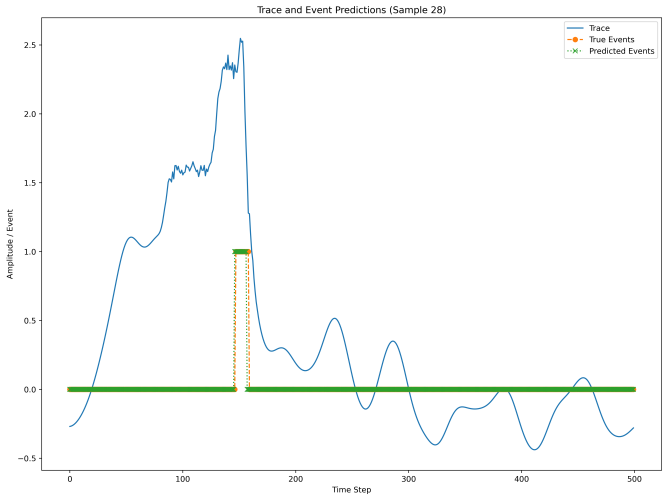

Figure 6: Windowed view of trace, prediction and ground truth

ing Support Vector Machine (SVM), Temporal Convolution Networks (TCN), Transformers, and Sequential Recurrent Neural Networks (LSTM). Each model was tested under various hyperparameter configurations, and the results were compared based on key performance metrics such as Precision, Recall, and F1 Score.

Among these models, the LSTM demonstrated the best performance, achieving an impressive F1 score of 0.92, which outperformed the other models in terms of its ability to accurately predict time-series events. To reconcile this, we can hypothesize models like Transformers are perhaps too sophisticated for our data — inter-sample interaction is far simpler and have shorter interaction ranges than the kind of complex data a transformer usually excels at, like token-interactions in language. Moving forward, further optimization and fine-tuning of hyperparameters, along with model ensemble techniques, may provide even more improvements in performance.

After a thorough evaluation of various models and data formats, we gained valuable insights into the performance of different architectures. Notably, our results indicate that a relatively simple model, such as the Long Short-Term Memory (LSTM) network, outperformed more complex networks, highlighting the importance of model selection and simplicity in achieving optimal results.

Model	Precision	Recall	F1 Score
SVM	0.675	0.666	0.670
Temporal Conv Network	0.874	0.784	0.827
Transformers	0.860	0.872	0.866
SequentialRNN	0.910	0.916	0.913

Table 2: Model Performance Comparison

References

Bai, S.; Kolter, J. Z.; and Koltun, V. 2018. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*.

Choi, K.; Fazekas, G.; and Sandler, M. 2016. Automatic tagging using deep convolutional neural networks. In *Proceedings of the 17th International Society for Music Information Retrieval Conference*, 805–811.

Chung, J.; Gulcehre, C.; Cho, K.; and Bengio, Y. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Lim, B.; Arik, S. O.; Loeff, N.; and Pfister, T. 2021. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting*, 1548–1555.

Malhotra, P.; Vig, L.; Shroff, G.; and Agarwal, P. 2015. Long short term memory networks for anomaly detection in time series. *Proceedings of ESANN*, 89–94.

Nadalin, J.; Storer, R.; Carmichael, D.; Freestone, D.; Grayden, D.; Burkitt, A.; Cook, M.; and Kuhlmann, L. 2021. Application of a convolutional neural network for fully-automated spike ripple detection in scalp EEG. *Journal of Neuroscience Methods*, 360: 109239.

Pachitariu, M.; Steinmetz, N.; and Kadir, S. 2016. Kilosort: real-time spike sorting for extracellular electrophysiology. *Journal of Neuroscience Methods*, 266: 1–15.

van der Molen, A.; et al. 2023. RT-Sort: A Convolutional Neural Network-Based Algorithm for Real-Time Action Potential Detection and Sorting. *PLoS ONE*, 18(3): e0312438.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is all you need. In *Advances in neural information processing systems*, 5998–6008.