HARVEY MUDD COLLEGE **Departments of Math and Computer Science**

Final Report for
Auburn / USDA

Time-series Modeling, Analysis, Interface, and Insight from Entomological Electropenetrography

May 8, 2025

Team Member

Mehrezat Abbas (Fall Team Lead)	mabbas25@cmc.edu
Devanshi Guglani	dguglani25@cmc.edu
Milo Knell	mknell@g.hmc.edu
Zachary B. Traul (Spring Team Lead)	ztraul@g.hmc.edu
Lillian Vernooy	ivernooy@g.hmc.edu

Advisor

Prof. Gabriel Hope

Liaisons

Dr. Elaine Backus
Dr. Anastasia Cooper
Dr. Kathryn Reif

Contents

Chapter 1

Project Overview

1.1 Background

Figure 1.1 *Aedes aegypti* mosquito feeding on a human host (Gathany (2006)). As the stylet is hidden inside the host's skin, it is necessary to use EPG to fully understand their feeding behavior.

Agricultural production in the United States is under growing threat from pests like sharpshooters, leafhoppers, and mosquitoes which act as vectors for diseases that decimate yields. As these pests spread these diseases when they feed on plants and livestock, understanding their feeding behaviors is crucial for creating targeted solutions. However, understanding their feeding is complicated by the fact that their mouthparts, formally called

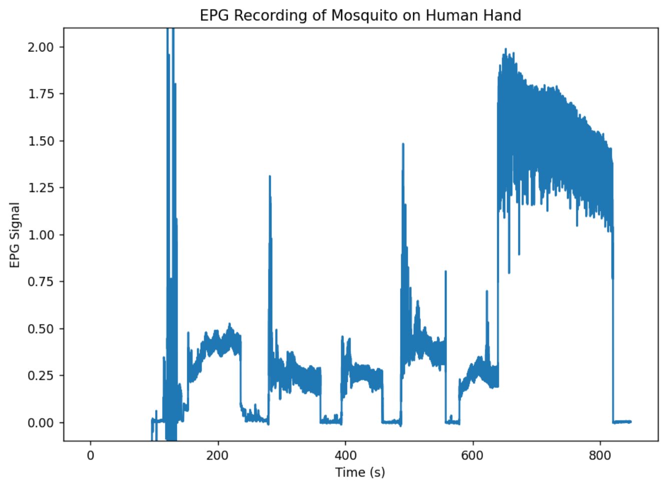


Figure 1.2 Example of an EPG recording using data from Cooper et al. (2024). While detail is hard to see on this scale, notice that it can be roughly broken up into five sections of higher voltage. These correspond to times when the mosquito had penetrated the host's skin.

stylets, are inserted inside of their hosts while they feed and therefore hidden from view (Figure 1.1). One solution to this observation problem is to use a technique called electropenetrography (EPG) to glean information about feeding behaviors.

At a high level, EPG generates data from feeding behaviors by placing a parasitic insect and its host in series with a power source and then measuring the voltage across the insect-host part of the circuit (Figure 1.3). As the insect feeds, its electrical resistance changes as a result of movement and penetration of host cells. This change in resistance then changes the voltage across the insect-host pair. Plotting that voltage against time yields an EPG recording (Figure 1.2). By studying these recordings in tandem with what can be observed while these pests feed, entomologists are able to characterize the appearance of an EPG recording during different feeding behaviors. These different appearances are referred to as "waveform families" in EPG literature (Cooper et al. (2024)). With the waveform families described, entomologists can study these pests at a larger scale without having to directly observe each insect because the EPG recordings can be analyzed after a given experiment. More concretely, this analysis consists of segmenting the time series of voltage values into segments of the same waveform family.

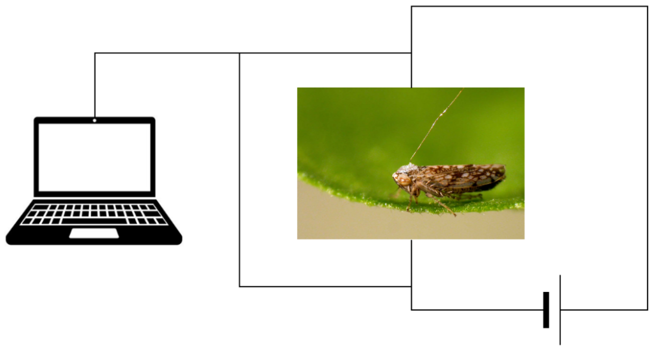


Figure 1.3 This figure depicts a simplified view of an EPG circuit. In it, we see a leafhopper with a gold wire glued onto its head on a host plant. The leafhopper and the plant are placed in series with a power source (Anonymous (2007)). We use measurements of the voltage drop across the insect and its host to generate an EPG recording.

This segmentation is currently done manually by our sponsors, a time-consuming process that takes away from other research. Thus, they seek to automate it. There have been many efforts to automate EPG segmentation in the EPG literature (Willett et al. (2016); Adasme-Carreño et al. (2015); Dinh et al. (2025)). While these algorithms achieve high levels of accuracy on aphid datasets, their performance is unknown on data from mosquitoes, our sponsor's focus. Some of these methods also make assumptions about recording appearance that are unique to aphids which renders them unusable on mosquito data. Finally, using their algorithms requires computer science expertise that renders them inaccessible to our sponsors. Therefore, our sponsors seek for us to evaluate automatic segmentation algorithms on a mosquito dataset and create software that makes these algorithms easy for entomologists to use.

1.2 Goals

With the above context in mind, our goal for the project was to create and evaluate a variety of algorithms for EPG segmentation and package them into a graphical user interface (GUI) capable of using them to segment EPG recordings and modify their outputs.

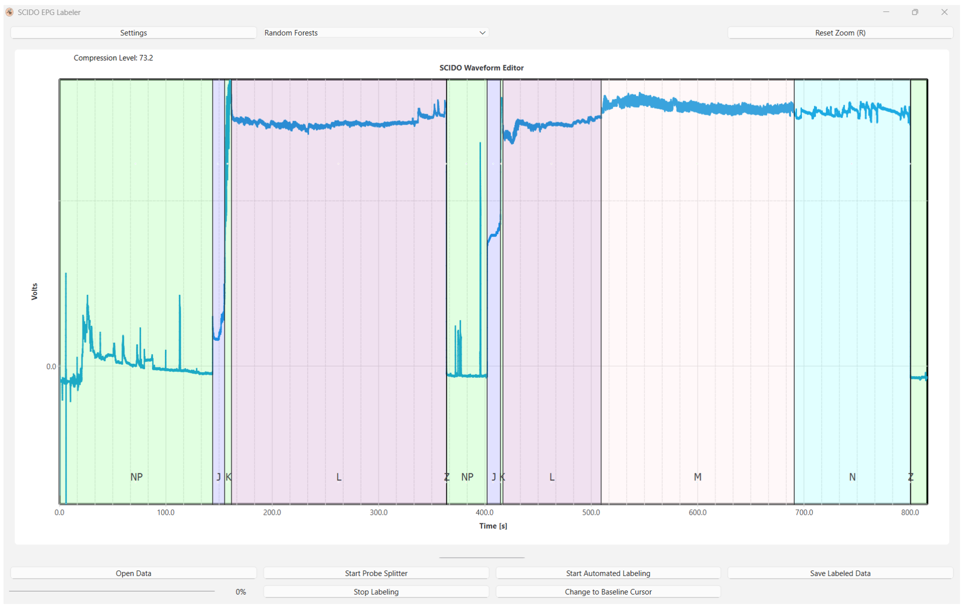
1.3 Accomplishments

1.3.1 EPG Segmentation Algorithms

This project resulted in the implementation and characterization of five algorithms for EPG segmentation which will be detailed in an upcoming paper from our liaison Dr. Cooper. The best of these algorithms achieved 84% accuracy on our dataset of mosquito recordings, a number we expect to increase as more data comes available. In addition, we developed algorithms for pre- and post-processing of EPG data that improve the performance and human usability of our segmentation algorithms. While the successful algorithms were important accomplishments, we also investigated several techniques that ended up being unsuitable for this task. Now that we have an improved idea of which techniques do and do not work, we can better inform future research in this area. For details on our segmentation algorithms, please see Chapter 3.

1.3.2 Graphical User Interface

This project also succeeded in creating a GUI whose capabilities go beyond just allowing a user to apply the above algorithms to an EPG recording. Our final application, called SCIDO (Supervised Classification of Insect Data and Observations), features a bespoke interface for interacting with recordings, customizable appearance, intuitive mouse-based controls, and tools for manual labeling (Figure 1.4). In addition, it is set up with future extensibility in mind. Even though our project's focus was EPG data from mosquitoes, integrating new algorithms and waveform types is easily done using our backend interface. For techical details on SCIDO, please see Chapter 4. For instructions on how to use SCIDO and install it for development and general use, please see the Appendix.


Random Forests

Compression Level: 73.2

Figure 1.4 This figure shows a screenshot of our completed software, SCIDO. Displayed in the software is an EPG recording with labels.

Chapter 2

The Data

2.1 Data Sets

Our primary data set consists of EPG recordings of *Culex tarsalis* (Western Encephalitis Mosquito) made on human hands, described in Cooper et al. (2024). The recordings are voltage time-series data in the form of WinDaq (`.WDQ`) files created using the WinDaq acquisition software. These are accompanied by CSV files and Excel spreadsheets which provide timestamps of transition points between waveform types for each file. They also provide other information about the recording configuration, such as the excitation voltage level and the input impedance of the amplifier.

WinDaq uses a binary file format, but the format is publicly documented on DataQ's website (https://www.dataq.com/resources/techinfo/ff.htm). In addition, there is existing code, in the form of the open-source windaq3 project (https://github.com/sdp8483/windaq3), which reads these files and returns the data in the form of a NumPy array. Using this library, we converted the WinDaq files into densely labeled CSV files to be able to work with them more easily. After cleaning, the *Culex tarsalis* data set consists of 62 recording files totaling 17 hours of recording. Of those 17 hours, 11 of them were labeled as "probing" behavior while the rest were non-probing. By "probing" we mean that the data in those segments came from when the mosquito was feeding. Towards the end of the fall semester a secondary data set of recordings of *Aedes aegypti* was also made available to us. After cleaning the data to include only files for which accompanying timestamped transitions were also available (which eliminated more than half the files), the data set consists of 24 files totaling 12.6 hours of recording. However, we opted to focus on *Culex tarsalis* and did not make significant use of this data set. We also have access to a set of recordings made on aphids, but we also did not make significant use of this data, as the focus of this project is mosquitoes.

Files recorded with the existing EPG system contain two signals, `pre_rect` and `post_rect`. The `pre_rect` signal is sampled from the output of the amplifier, and the `post_rect` signal is additionally passed through a rectifier and low-pass filter. The dataset consists of a mix of recordings made with DC and AC excitation signals, and the AC excitation signal is at 1 kHz. Both `pre_rect` and `post_rect` are sampled at 100 Hz, meaning the Nyquist frequency is only 50 Hz. In the AC recordings, we noticed the `pre_rect` signal contains a low-amplitude oscillation at around 10 Hz to 20 Hz which is not present in `post_rect`, and this is presumably the excitation signal being aliased down. For the `post_rect` signal, the low-pass filter serves as an anti-aliasing filter, filtering out the excitation frequency before the signal is sampled. This is shown in Figure 2.1.

For most of the files in the tarsalis dataset, the baseline of the `pre_rect` signal is at around 0 V, and the `post_rect` signal is shifted up by between 5 V and 8 V relative to the `pre_rect` signal. Both signals appear to clip at 10 V, and since the `post_rect` signal is shifted up, there are instances where `post_rect` clips while `pre_rect` does not. An instance of this is shown in Figure 2.2. (Since both signals clip, and at the same voltage, the clipping is probably caused by exceeding the ADC input limit, rather than some issue with the amplifier, rectifier, or low-pass filter).

So, there are tradeoffs between using the `pre_rect` signal and `post_rect` signal. While the amplitude is relatively small, the aliased excitation frequency in `pre_rect` might obscure the waveform signals we are interested in, especially if we are interested in components at a similar frequency. On the other hand, the `post_rect` signal clips more frequently than the `pre_rect` signal. In practice, we didn't find a very significant different in the machine learning performance between the two, and so we used each in different cases.

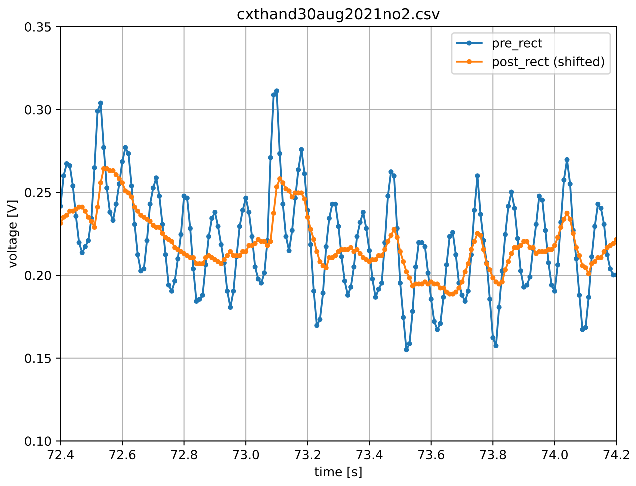


Figure 2.1 The `pre_rect` signal, showing the excitation signal aliased at about 11 Hz, and `post_rect` with the excitation signal filtered out. The `post_rect` signal has been shifted to the same DC level as `pre_rect` for visualization purposes.

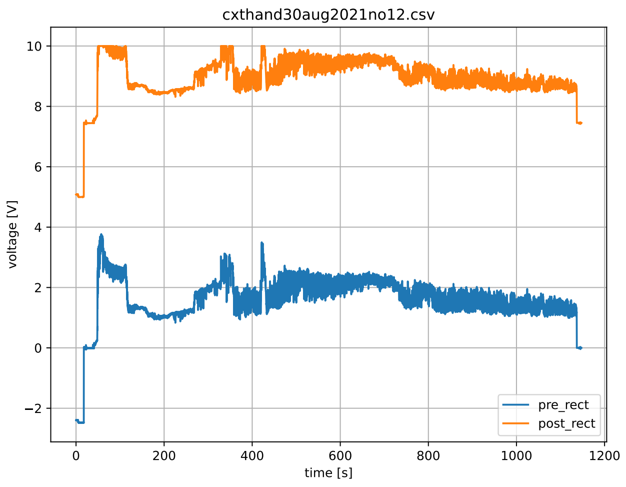


Figure 2.2 A complete recording showing the `pre_rect` signal (not clipping) and the `post_rect` signal clipping at 10 V.

2.2 Preprocessing

We also performed an additional preprocessing step of "probe splitting." Probing (P) is when the insect has inserted its mouthparts into the host, and probing regions of the recording are characterized by a higher voltage level than baseline/nonprobing (NP). A given file generally begins and ends with nonprobing, with probing in between, and a given recording file might contain multiple probes if the insect removes and re-inserts its mouthparts. We often found it helpful to split the recording into regions of continuous probing, and examine these individually. We used a simple procedure — first, apply a moving average (in our code we used a default window size of 500 which corresponds to 5 seconds). Then, treat regions above a threshold as probing, and below as nonprobing (in our code we used a default threshold of 0.1 V). Finally, only include probes of a certain length (default 1500 samples, or 15 seconds), and include some number of NP samples on either side of the probe (default 500 samples, or 5 seconds).

Chapter 3

Machine Learning Models

In this chapter we give a detailed description of the machine learning models and algorithms we investigated during this project, including models which we did not implement in SCIDO.

3.1 Segmentation Algorithms

All of the following models were hyperparameter tuned using Optuna, developed by Akiba et al. (2019), to automatically tune for 100 iterations.

3.1.1 Random Forests

Random forests are an ensemble model consisting of a collection of decision trees, each trained on a bootstrapped sample of the training data and grown by using a random subset of the available features for each split. In a classification setting, an input is classified by feeding it into each of the decision trees in the ensemble and returning the class predicted by the most decision trees. Although individual decision trees have high variance, by de-correlating them during training and ensembling their outputs we get a flexible model without too much variance. As random forests often perform well out of the box without much additional effort, they are a

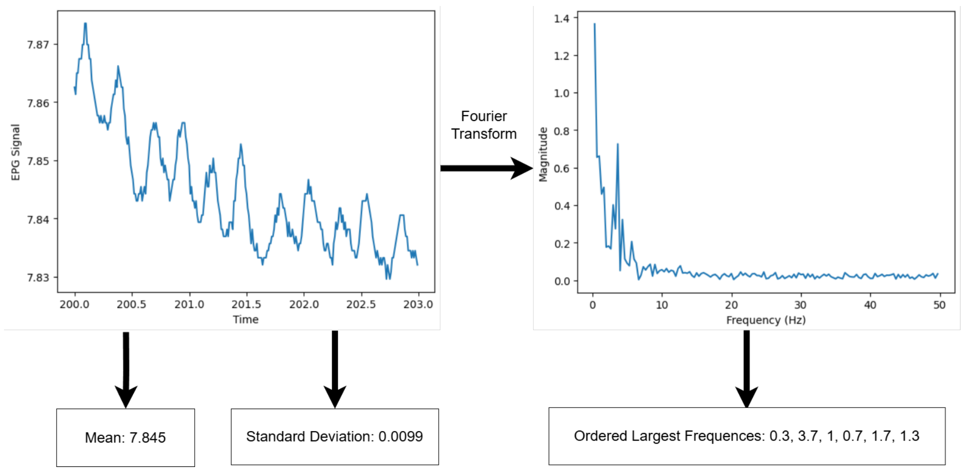

EPG Signal
Time
Fourier Transform
Magnitude
Frequency (Hz)
Mean: 7.845
Standard Deviation: 0.0099
Ordered Largest Frequences: 0.3, 3.7, 1, 0.7, 1.7, 1.3

Figure 3.1 Feature extraction used in the random forests method. From the raw EPG signal (left), we extract the mean and the standard deviation of the window, which is 3 seconds long in this case. From its Fourier Transform (right), we extract the frequencies with the largest magnitude, in order.

popular initial choice for a variety of machine learning tasks (Hastie et al. (2009)).

We follow the work of Willett et al. (2016) in applying random forests to EPG and frame our EPG probe segmentation task as a supervised classification task on non-overlapping, fixed-width windows of the unrectified probe signal where the target class of each window is the waveform type with the longest duration in that window. Features extracted from each window consisted of the mean and standard deviation along with the frequencies whose magnitudes were the largest after applying a Fourier transform to that window (Figure 3.1) . These values allow us to characterize the signal appearance in each window, which can consist of hundreds of values depending on the window size, in fewer values which can then be used as features in a random forest. Additionally, as the appearance of a waveform type is known to vary based on the settings used during EPG recording, we also pass in the impedance, voltage, and current type (alternating or direct) used on the EPG recording device while the signal was being measured. In this case of our dataset, this was a four-channel AC–DC electropenetrograph built by Andrew Dowell, where settings were varied across a range of values as described in Cooper et al. (2024).

The implementation of feature extraction and the random forest models were respectively done in using the Python libraries scipy and scikit-learn as they provided easy out of the box functionality to implement this method (Virtanen et al. (2020), Pedregosa et al. (2011)). For feature extraction, we used scipy's fft function to compute the Fast Fourier Transform of each window. For the random forest model, we used scikit-learn's RandomForestClassifier. Based on hyperparameter tuning, we achieved our best results using windows of size 3, extracting 7 frequencies per window, and using 128 estimators with a maximum depth of 16 for the random forest model, which is the architecture implemented in SCIDO for the random forests model in the file `rf.py` in the repository in the project archive.

3.1.2 Segmentation Transformer

The Segmentation Transformer is an attention-based encoder-decoder architecture originally developed for image segmentation which we modified for time-series data (Zheng et al. (2020)). At a high level, this architecture segments by dividing its input into "patches", embedding them, passing them through an attention encoder, and then using a series of convolutions and upscales to decode (Figure 3.2). This architecture is inspired by Vaswani et al. (2023) and is notable in that it does not make use of convolution in its encoder. As this architecture is attention-based, we chose it with the expectation the wide receptive field would improve performance compared to methods like random forest which have an incredibly narrow receptive field. Based on discussion with our liaisons, we believed being able to capture long-range relationships could improve performance on our minority waveforms like N.

Our specific implementation is similar to the Progressive Upscaling variant described in Zheng et al. (2020), with our modifications being necessary to use their architecture designed for 2-dimensional images on a 1-dimensional time series. As our inputs were not fixed-width we divided it into fixed-width windows, the number of which depended on the length of the recording, instead of a fixed number of patches. Additionally, we only used 1-dimensional positional embeddings and convolutions. Instead of using a bilinear upscaling, we used a linear one. Finally, because the length of our input was variable we had a variable number of upscaling steps that depended on the length of the input instead of a fixed number.

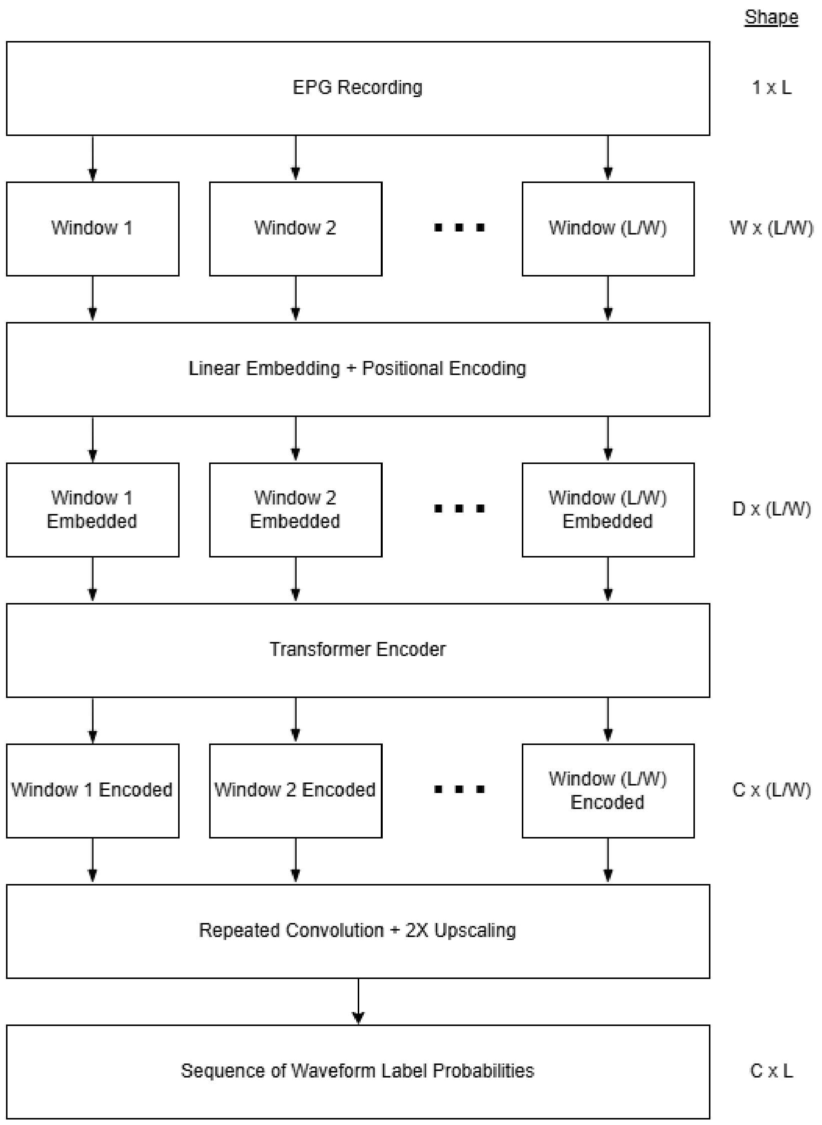


Figure 3.2 This figure shows the architecture of the Segmentation Transformer model, starting with the raw EPG recording on top and ending with labels on the bottom. L is the length of the original recording, W is the width of each window, D is the embedding dimension, and C is the number of labels.

We implemented this model using PyTorch as it is industry-standard for this task and it implements most of the components we needed to create and train the model (Paszke et al. (2019)). From hyperparameter tuning, we found that using an embedding dimension of 32 and a transformer encoder with 2 layers and 16 heads gave the best performance. For training, we achieved best results using the Adam optimizer with a learning rate of 0.0005 for 32 epochs. These settings were used for the Segmentation Transformer model included in SCIDO in the file `transformer.py` in the repository in the project archive.

3.1.3 Fully Convolutional Network (FCN)

Our method was initially based on a Temporal Convolutional Network (TCN) as developed by Bai et al. (2018), which work to solve the main problem with traditional convolutional neural networks: small receptive field. The receptive field is how much of the input data the model has access to at each data point. Traditional CNN stacks have a receptive field that grows slowly (linearly) with depth, while TCNs use dialted convolutions. This allows each layer to skip some timestamps, meaning that the receptive field can grow exponentially with the size of the network, similar to a UNet.

The network consists of a stack of these convolutions, with the dilation increasing each time by some dilation base so the i-th layer will have a dilation of dilation_base^i (traditionally with dilation_base = 2). Each subsequent layer incorporates information from a wider and wider view of the data. There are also skip connections between layers to pass the information from the low-level view of the data to higher-level views.

The wide receptive field and ability to incorporate information from a wide range of data makes this a good choice for our task. Traditionally, TCNs only use past information to predict the future which makes sense in contexts like finance (called causal convolutions). However, in our case, looking at the future data is not only acceptable but required for a full picture. Thus, a fully convolutional network (FCN) as developed by Long et al. (2015) is this causal convolution, making FCN a more technically correct name. However, FCNs are a more general architecture and we wanted to acknowledge the inspiration we took from the more specific TCN.

The model makes predictions on non-overlapping windows and combines information from before and, of a specified length as a hyperparameter.

We implemented this model using PyTorch as it is industry-standard for this task and it implements most of the components we needed to create and train the model (Paszke et al. (2019)). From hyperparameter tuning, we found that channels growing following [52,64,96,144,216,324], a kernel size of 5, dropout of 0.001, 8 epochs, dilation base of 2, 3 seconds of data before, a half second data window, 3 seconds after. These settings were used for the FCN model included in SCIDO in the file `fcn.py`.

3.1.4 UNet

UNets, so called for the shape of their architecture (Figure 3.3) were originally developed for biomedical image segmentation by Ronneberger et al. (2015). This architecture works by downsampling (halving the size of the input size) the image repeatedly, called the encoding path, to capture context and then upsampling (doubling the size of the image again), called the decoding path, back up to the original resolution of the input image. See Figure 3.3 for an example of what a UNet looks like (although this one is for 2-dimensional images rather than 1-dimensional time series).

Information is transferred across the U (grey arrows) and also between layers (not pictured in this image). This flow of information, called skip connections, can help the model to consider information at many levels of abstraction.

The (en/de)coding paths both consist of blocks, followed by either an (down/up)sample denoted by (red/green) arrows in the diagram. Each block consists of two convolutions (recall, a convolution allows the network to apply a learned filter to the data) followed by some other steps to clean up the output (normalization and activation, plus potentially dropout). The blocks allow the network to process the information at that level of abstraction, and then it is (down/up) sampled to be processed at the next level of abstraction.

At the lowest point of the U (called the bottleneck), we can either apply another block or use a more complex transformer model. The transformer, developed by Vaswani et al. (2023) was originally developed for processing text. It is powerful because it allows the model to look at entire sequence

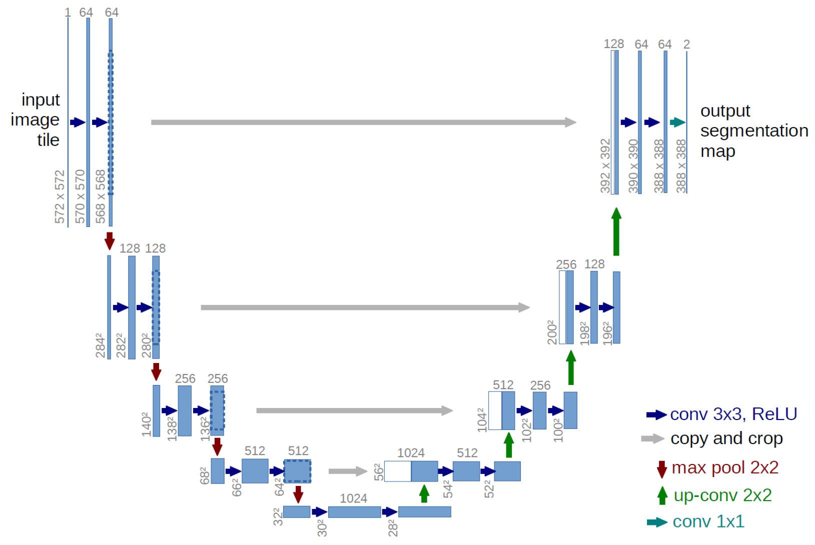


Figure 3.3 This figure, produced by Ronneberger et al. (2015), gives an example architecture of a UNet on 2-dimensional images.

at once using a mechanism called self-attention. However, even after being downsized through the encoding path of the UNet the data is still too large for a transformer to be run in a reasonable time. To remedy this, we use windowed attention (also sometimes called sliding window attention), similar to the implementation from Beltagy et al. (2020). This allows a UNet with windowed attention to be similarly fast to a UNet with a more standard block in the bottleneck.

UNets were originally designed for segmentation, which is very similar to the waveform labeling problem, giving them a strong theoretical reason to perform well. They are a particularly good choice for our task since they can look at the entire input sequence, avoiding some of the issue present in windowing like abruptly cutting our data. They offer an exponentially growing receptive field, similar to a TCN/FCN, and are reasonably fast to run.

Our implementation differs from the standard UNet primarily because our problem is a 1-dimensional time series, as opposed to the 2-dimensional images UNets were originally designed for. For this reason, we recommend a growth factor of 1 both theoretically and empirically, as opposed to the more standard 2 in image segmentation problems.

We implemented this model using PyTorch as it is industry-standard for this task and it implements most of the components we needed to create and train the model (Paszke et al. (2019)). From hyperparameter tuning, we optimized for both the block UNet and the variant with attention in the bottleneck.

For the UNet without attention, we find that the best model has:

```
{'epochs': 64, 'lr': 0.0005, 'dropout_rate': 0.1, 'weight_decay':
    1e-06, 'num_layers': 8, 'features': 32}
```

For the UNet with attention, we find that the best model has:

```
{'epochs': 64, 'lr': 0.0005, 'dropout_rate': 1e-05,
    'weight_decay': 1e-05, 'num_layers': 8, 'features': 64,
    'transformer_window_size': 150, 'transformer_layers': 2,
    'nhead':2}
```

These settings were used for the UNet model included in SCIDO in the file `unet.py`.

3.2 Evaluation and Results

All of the above models were evaluated on the entire dataset from Cooper et al. (2024) for their accuracy and macro F1 score using 5-fold cross validation. To avoid data leakage from multiple probes by the same mosquito in the same recording, we obtained our folds by splitting on a recording level as opposed to a probe level. Accuracy was calculated as the number of correct predictions across all 5 folds of probes divided by the total number of predictions. That is,

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Predictions}}.$$

The macro F1 score was obtained by first averaging the F1 score of each model on each class. The F1 scores on each class were in turn calculated

Model	Accuracy (%)	Macro F1 Score
Random Forest	77	0.39
FCN	79	0.37
Segmentation Transformer	74	0.39
UNet (block)	**84**	0.66
UNet (attention)	82	**0.69**

Table 3.1 Accuracy and macro F1 score for each model. Values in bold indicate the highest performance on each metric.

from their precision and recall scores as follows:

$$\text{F1 Score}_c = 2\frac{\text{Precision}_c \cdot \text{Recall}_c}{\text{Precision}_c + \text{Recall}_c}$$

where $c \in C$ is a class from the set of classes C and

$$\text{Precision}_c = \frac{\text{True Positives}_c}{\text{True Positives}_c + \text{False Positives}_c}$$

$$\text{Recall}_c = \frac{\text{True Positives}_c}{\text{True Positives}_c + \text{False Negatives}_c}.$$

Therefore, the macro F1 score is given by

$$\frac{1}{|C|}\sum_{c\in C}\text{F1 Score}_c.$$

We chose to evaluate accuracy because it is the most common measure of performance in EPG literature. However, due to the imbalance of our dataset and the limited information given by accuracy we also decided to evaluate the models based on their macro and micro F1 scores. In particular, we did not use a weighted F1 score because our liaisons wanted each waveform to be valued equally during evaluation. These values for each model are summarized in Table 3.1.

In addition to the accuracy and macro F1 score, we also calculated per-class F1 scores (Table 3.2). Confusion matrices and plots of actual vs predicted labels for every probe can be found in the `drcooperreport` folder in the project archive. The code used to generate this report can be found in the file `model_eval.py` in the compressed code repository in the project archive.

Model	J	K	L	M	N	W
Random Forest	0.47	0.19	0.78	0.80	0.05	0.00
FCN	0.40	0.06	0.80	0.89	0.04	0.00
Segmentation Transformer	0.70	0.11	0.77	0.77	0.00	0.00
UNet (block)	0.87	0.60	**0.85**	**0.91**	0.14	0.59
UNet (attention)	**0.88**	**0.67**	0.83	0.87	**0.32**	**0.62**

Table 3.2 Per-class F1 score for each model and label. Values in bold indicate the highest performance on each class.

Although our best model's accuracy of 84% is is lower than results of most EPG papers which claim accuracies of greater than 90%, we believe this is nevertheless a strong result in the context of our data and the way we chose to evaluate our models. With respect to our data, we have much less of it than in other studies like Willett et al. (2016), which had 40 times as much training data as we did. Additionally, we chose to exclude the easily-labeled NP regions that make up more than a third of our pre-filtered data from our evaluation. If we were to include them in our final statistic, our best accuracy would climb to 90%. However, adding NP regions to our evaluation would be misleading when segmenting behaviors within probes is our project's focus.

Looking at the results in Tables 3.1 and 3.2, our best models are clearly the two variants of the UNet. In particular, the F1 score of the UNet is much higher than the other models on waveform types like J, K, N, and W. We hypothesize that its improved performance on J, K, and W is because it is the only model which does not divide inputs into fixed-width windows and then assign the same label to every time point in each window. As waveform types J, K, and W are short, they often do not make up the majority of any window and thus the assigned label for that window often does not correspond to them. Beyond its fine-grained output, the UNet is a good theoretical choice due to its original purpose of and success in image segmentation.

Although the fine-grained approach that the UNet takes produces better results by these metrics, it also leads to noisier results. In particular, it creates a phenomena we call "barcoding" for the appearance of the rapid alterations between two labels when plots like those in Figure 3.4 are viewed at higher zoom levels. For a human user, barcoding is a serious problem because manually removing them from a single recording would require hundreds of mouse clicks in SCIDO or any other software. Therefore we also looked into post-processing techniques that could alleviate this problem.

3.3 Postprocessing

As shown above, barcoding is a serious problem for our liasons. We have considered several methods for fixing this problem. In Figure 3.5, we demonstrate the effect of postprocessing on a single probe.

3.3.1 Barcode Deleter

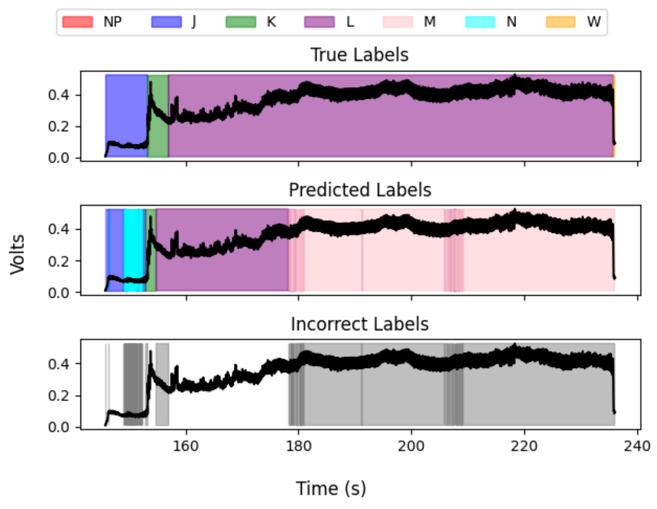
The Barcode Deleter is a naive method that does not make use of any training data. It removes barcodes by scanning the file from beginning to end and replacing any labels that are too short for their waveform type with the labels of the segments adjacent to that short segment. The minimum segment length for each waveform type is set individually. Although simple, it runs in $O(N)$ time where N is the length of the probe and it is guaranteed to remove repeated short segments that are tedious to remove manually.

3.3.2 Smoother

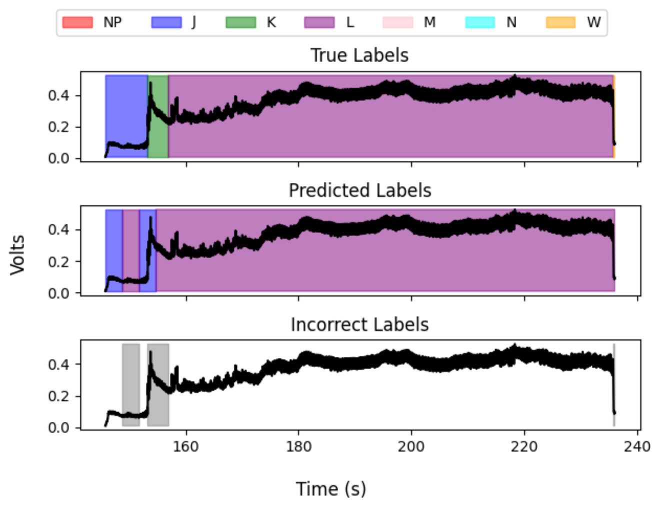
By applying a Savitzky-Golay filter, as designed by Savitzky and Golay (1964), we can smooth the logits produced by the model into a less noisy distribution. In Figure 3.5, the effect of the filter is evident as it removes some small spikes of other labels that would create small, few tick, barcodes. This approach is simple and fast to run, but still leaves many barcodes in place and is not a substitute for a more comprehensive method.

3.3.3 HMM

A Hidden Markov Model (HMM), as described by Degirmenci (2014), can also be used to solve this problem. We can take the logits produced by the model and apply a softmax activation to produce a probability distribution.



a. UNet Labels



b. Random Forest Labels

Figure 3.4 A comparison of the outputs between UNet (a) and Random Forest (b) on the same probe. While there are differences in predictions, the difference of note is that the UNet's predictions are noisier because it is more fine-grained.

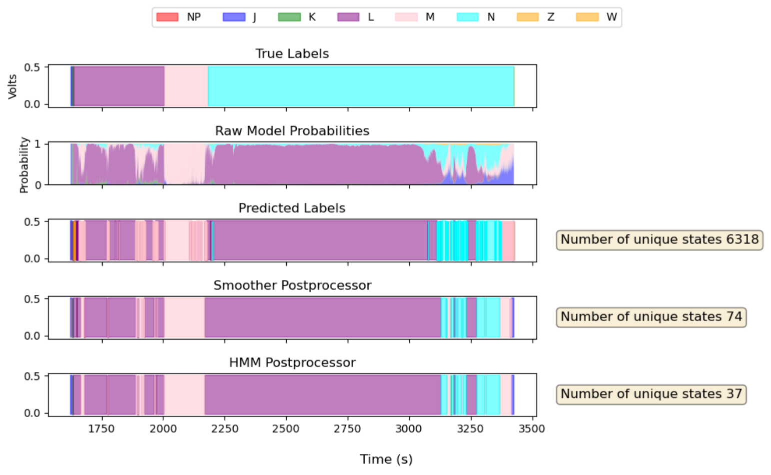


Figure 3.5 A single probe was randomly selected from our test set. Then, we run the best UNet with attention and extract the logit probabilities at each tick. We display those probabilities, as well as the predicted class at each tick. We then separately apply each of our post-processing methods to this probe and show their output. Finally, for each output we count the number of unique states which can be used as a proxy for the number of barcodes. This counting step is also useful because it can be hard to visually notice a single tick barcode.

We then have a distribution over labels at each tick, and we can treat this as the emission probabilities (i.e the probability of observing a label if the model is in that state). This allows us to solve this problem by leveraging the structure of an HMM, without building one ourselves and instead relying on a more sophisticated model such as the UNet (as pictured in fig 3.5). Once we have these emission probabilities, we can run a standard HMM decoding step using the Viterbi algorithm Forney (1973), which runs in $O(N)$, making it very efficient.

The HMM post-processor creates valid recordings by following a transition matrix passed as input. This can either be empirically generated from training data or taken from a paper, and then optionally modified to allow traditionally illegal transitions to make it more flexible when dealing with non-standard data. This might be useful when viewing recordings where a pesticide treatment has been applied to the insect, which can cause transitions normally considered illegal in a healthy insect.

This is our recommendation for the best working postprocessing algorithm.

As can be seen in Figure 3.5, this produces the fewest barcodes, and of those that do occur they only really happen between M and L. This is because a traditional HMM treats the time in each state as following an exponential distribution, making it memoryless. This is empirically not a true assumption, we found that all states follow an approximately normal distribution for dwell times. There may be ways to fix this within the framework of a traditional HMM, such as applying the barcode deleter to remove the small transitions between M and L.

3.3.4 HSMM

The theoretically correct way to add memory to the time spent in each state is with a Hidden Semi-Markov Model (HSMM). This model, as described by Murphy (2002), works by having an explicit function for the probability distribution over time spent in each state. This is a great choice for our problem, since we can pass in the explicit normal distributions using parameters found from the data.

The main problem here is runtime. Decoding for a HSMM takes $O(N^2)$, making it difficult for our data where N can be in the hundreds of thousands. Preliminary benchmarks suggest that our implementation could take a few hours to run the HSMM on a single probe, which is unreasonable for our liaisons. There is much room for improvement in the implementation of the algorithm by using parallelism and other runtime optimizations, since this was not an area we explored in sufficient depth.

We do not have a working end-to-end implementation of the HSMM, so we cannot display its output in Figure 3.5. However, we believe this is the most theoretically correct way to solve the problem and makes barcoding nearly impossible because observing short times spent in waveform types M and L is very improbable under the normal distributions we found.

3.4 Other Approaches

In addition to the supervised methods described above, we also investigated using unsupervised segmentation algorithms as a pre-processing step to either provide another channel of information to our models or as a way of first segmenting the waveform before applying a label to each of the segments with another model. In particular, we investigated using the ClaSP algorithm from Ermshaus et al. (2023), Hidden Markov Models for classification, and various change-point detection algorithms implemented in the Python library ruptures Truong et al. (2020). We hypothesized that these methods and particularly the change-point detection methods would better emulate how humans manually segment waveforms. While these methods initially appeared promising on a small sample of recordings, they did not perform well in general. One reason for their poor general performance is that they required knowledge of the number of expected states or change points in a given probe beforehand. As we could not know this in general, we would have to either overestimate or guess the number of states or change points for each probe. We also attempted to use the intermediate values in the ClaSP algorithm as inputs to some of our models, but we found that they either had no effect on performance or they decreased it.

In addition to the above unsupervised algorithms, we also attempted including a Conditional Random Field in the final stage of some of our models. If successful, it would have allowed us to encode the probability of transitioning from each waveform type to the next into the model itself. This is similar to what some of the post-processing techniques above do. However, we found that adding a Conditional Random Field to our models made them incredibly slow to train and we were unable to characterize their affect on performance.

Finally, one model-agnostic technique we attempted to incorporate into training was data augmentation. As our dataset was somewhat small, we hoped this could help reduce overfitting and allow us to use deeper architecture. While there were many ways we could have attempted to augment the data, the method we fully evaluated consisted of scaling the data in both the time and voltage domains and mixing waveform types from different files. More specifically, building one probe in our augmented dataset consisted of repeatedly sampling one segment from a random time- and voltage-scaled probe in the training set to build a "franken probe". That is, we would first sample a J waveform, then a K waveform, L or M waveforms, and so on, until we sampled a W waveform. At each step, the next probe in the sequence being determined probabilistically according to the transition matrix obtained from the training data. In this way we build an augmented dataset from our training data. Although we think data augmentation would be useful if given more time, in our testing it did not improve performance.

Chapter 4

Graphical User Interface

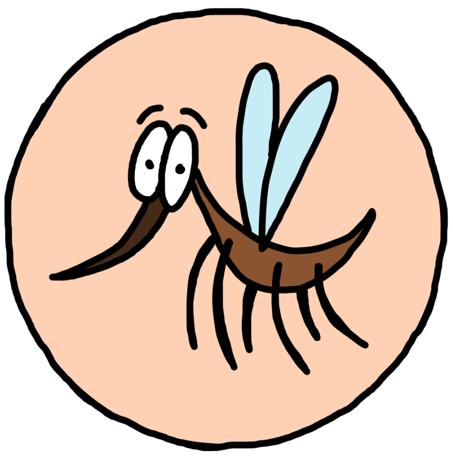


Figure 4.1 Skeeto, the official mascot of our software product!

In this chapter we describe the interface components of the SCIDO EPG Labeler software, including user-facing features and developer-friendly class documentation.

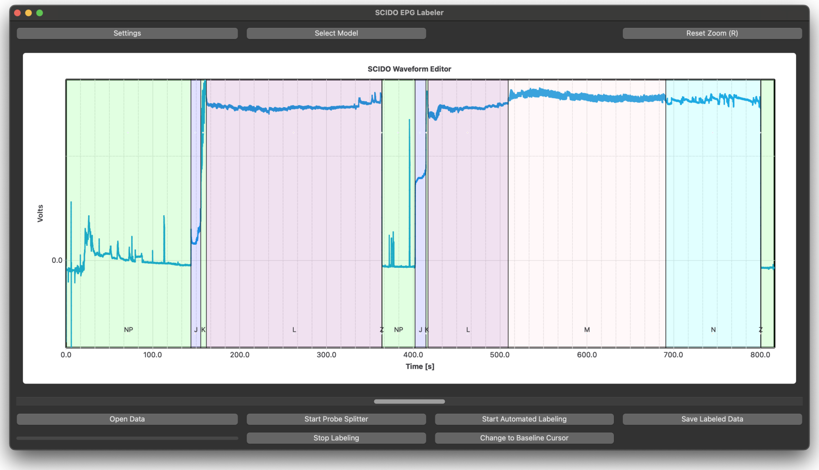


Figure 4.2 Data Window running default settings. This is what the user is presented with upon opening the SCIDO application.

4.1 Features

The SCIDO software was created to allow users to interface with our machine learning labeler model. It contains controls for running the model, as well as tools for manual correction of the labeled data. The complete set of possible actions and features is listed in the following subsections.

4.1.1 Opening Data Files

To open a new file for labeling, click the `Open Data` button in the bottom left corner of the window. Navigate to your desired CSV or WINDAQ file and open it. The data and labels from this file will be plotted in the data window.

4.1.2 Manual Label Adjustment

There are four forms of label modification available in the data window:

- Click and drag existing transition lines to move them.
- Right click on existing transition lines to delete them.
- Right click between transition lines to add new transitions.
- Double click between transition lines to change existing labels.

These modifications can be performed on both probe and label identifications.

4.1.3 Data Chart Navigation

In the chart view, we have implemented both scrolling and zooming capabilities:

- Scroll on a mouse wheel or mouse pad to scroll horizontally.
- Hold `SHIFT` while scrolling on a mouse wheel or mouse pad to scroll vertically.
- Hold `CTRL` while scrolling on a mouse wheel or mouse pad, or perform the zoom gesture on a mouse pad to zoom horizontally.
- Hold `CTRL` and `SHIFT` while scrolling on a mouse wheel or mouse pad, or hold `SHIFT` while performing the zoom gesture on a mouse pad, to zoom vertically.
- Click the `Change to Baseline Cursor` button at the bottom of the screen to toggle the cursor type. While in baseline mode, any clicks in the chart area will add a gray horizontal line to the chart at the y-coordinate of the click. This line can be removed from the settings window. To return to the normal cursor, click the button again.

Occasionally, after scrolling or navigating menus, the user must click somewhere inside of the chart before the zoom functionality will work as expected.

4.1.4 Probe Segmentation

To start the probe splitting step, which is necessary before running the automated labeler, click the `Start Probe Splitter` button at the bottom of the window. The software will automatically identify unique probes and display them in the data window. At this point, the user should assess the accuracy of these results and adjust them as necessary before starting automated labeling.

4.1.5 Selecting the Active Model

Our software comes with five model choices: Random Forests, UNet (Block), UNet (Attention), SegTransformer, and TCN. To select which model will be used during the next round of automated labeling, the user can click the dropdown at the top of the window.

4.1.6 Adjusting Visual Settings

There are a variety of visual settings that can be changed within the application according to user preference. These include:

- Add Label
- Change Label Color
- Change Line Color
- Toggle Chart Component Visibilities
- Adjust Gridline Spacing
- Adjust Gridline Offset
- Remove Baseline

These settings can be accessed by clicking the `Settings` button in the top left corner of the window. Selected settings can be saved by clicking the `Save settings` button at the bottom of the settings window, and will consequently be loaded each time the SCIDO application is opened.

4.1.7 Automated Labeling

To run the labeling model, click the `Start Automated Labeling` button at the bottom of the window. The model will run, and its results will be displayed in the data window. At this time, the user should examine the labels and adjust them for accuracy.

4.1.8 Exporting Labeled Data

Transitions and labels can be exported as CSV or plaintext files by clicking the `Save Labeled Data` button in the bottom right corner of the window. To save the file, go to the desired file directory, give the file a name, and click `Save`.

4.2 Tools

The GUI software is written in Python due to its simplicity and ability for rapid iteration compared to other languages. In addition, writing our GUI in Python made it easy to integrate with our machine learning code which was also written in Python.

We used the PyQt library to create the interface itself. PyQt consists of bindings to the C++-toolkit called Qt, which is a mature framework with a wide range of features from a large widget set, robust layout management, and support for complex event handling. This made it ideal for implementing our interactive data labeling. Qt also enabled us to create a custom GUI for waveform visualization, labeling, and adjustment.

4.3 Classes

In this section we describe the functionality of each class and the relationships between the separate components of our codebase. This accompanies code comments as a means of informing new developers of the high-level functionality of the software.

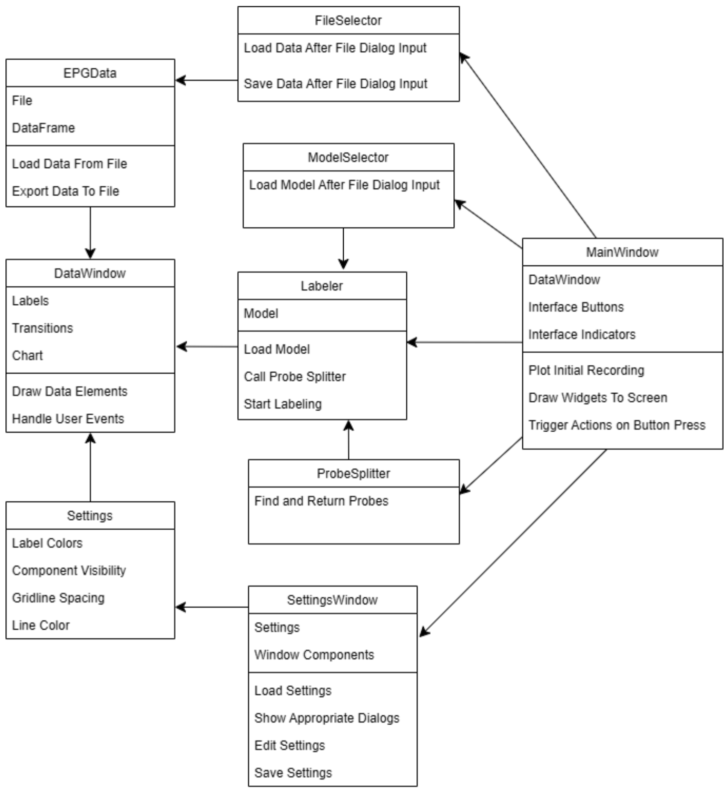


Figure 4.3 A UML class diagram representing SCIDO's structure behind the scenes. More detailed class descriptions can be found in the text below.

4.3.1 Main Window

The `MainWindow` class is where all of the on-screen widgets in the SCIDO application are created. This includes the buttons, progress indicator, and even the window icon. Each button is set to call a corresponding function, which means that most of the functionality within the rest of our classes is called in some manner from this class.

4.3.2 Data Window

The `DataWindow` class provides the main interactive chart component of the SCIDO software. It extends the QChartView-widget from the Qt library.

Upon initialization, it draws the chart area, axes, and example data. When new files are selected for display, this class is called to plot the new EPG data and redefine the tracked data for manipulation during probe splitting and labeling.

The `DataWindow` class also handles all manual data adjustments. When a user input such as a click, double click, mouse movement, or scroll occurs, the class determines the user's intended action and target. When transitions are moved, created, changed, or deleted in this manner, the class updates both the visual elements of the chart (including transition lines, label text, and duration display) and the appropriate data structures.

Chart navigation is also controlled by the `DataWindow` class. With scrolling mouse gestures, users can pan and zoom horizontally and vertically to adjust their view of the data.

Much of the Data Window's functionality–particularly transition events– were produced without support from PyQT. One of the greatest portions of effort towards the final software product was spent producing functions for manually tracking the positions of elements on the screen.

Figure 4.2 depicts the Data Window with default settings.

4.3.3 EPG Data

The `EPGData` class handles all file interactions within the SCIDO software. It is called to load data from a given CSV or WINDAQ file into memory as a Pandas DataFrame object. The class also contains functions for editing the labels and transitions stored in the DataFrame as they are manipulated from the Data Window. Finally, the `EPGData` class is responsible for exporting labeled and edited data as either a CSV or plain text file and saving to disk at the chosen location.

4.3.4 Probe Splitter

The `ProbeSplitter` class contains one function. This function, `simple_probe_finder`, executes the probe splitting algorithm as described in section 2.2. It is called from the Labeler class, whose `start_probe_splitting` function is triggered by a click of the "Start Probe Splitting" button in the SCIDO interface.

4.3.5 Labeler

The `Labeler` class controls the interactions between the loaded data and the machine learning-based labelers. When the user clicks the "Start Automated Labeling" button, the `start_labeling` function passes the data into the active model and returns the result. The probe splitting function described above is also called from the `Labeler` class.

This class also handles model selection. When a model (Random Forests, UNet (Block), UNet (Attention), SegTransformer, or TCN) is selected from the drop-down menu, the `Labeler` class loads the appropriate model to be used during subsequent automated labeling.

4.3.6 Model Selector

The `ModelSelector` class manages the selection and loading of machine learning models used for automated EPG labeling. Interfaces with model scripts (e.g., `rf.py`, `tcn.py`, `unet.py`) and loads the corresponding classification logic. It also links the model selection process with the automated labeling engine. This selector connects with the core labeling logic to define which ML model is applied during automation.

4.3.7 File Selector

The `FileSelector` handles file input output operations for waveform data and labeled outputs. It supports CSV and Windaq formats and controls data parsing, populating the main display window with waveform and label overlays, and managing save functions for post-labeling export. It also initializes and updates the primary data canvas, coordinating with labeling and settings systems.

4.3.8 Settings

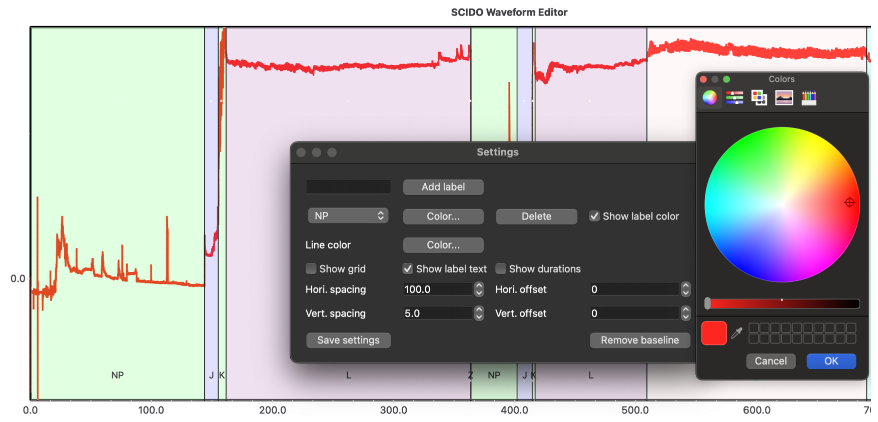
Our `Settings` class contains runtime-adjustable visual and semantic settings. Supports label addition, color assignment, deletion, and toggle of label visibility. It also includes waveform color changes, grid toggles, and spatial adjustments for grid lines, and generally control over the graphical fidelity of data representation. alters display and label behavior, feeding updates to the GUI and data visualization components. It should be noted that this class is such that the state of each setting parameter is stored in one place from which all components can read, rather than stored in the current state of several widgets which must be synchronized. As long as all the components are notified of a change in this class, they can read from it and update internally. This is somewhat easier to manage than an everything-to-everything connection between widgets, as it makes the checklist for testing much longer than it needs to be.

An example of this are the number of active labels in the system at present, which can be deleted, added and renamed. The data structures that store these are runtime statics in the Settings class, such that the Settings Window can write and the Data Window can read these more complicated types.

4.3.9 Settings Window

This GUI panel hosts sliders and buttons for adjusting the settings, primarily affecting the visualization in the data window. It is effectively an interface layer for accessing and manipulating visual and label configuration options. It extends the generic QObject-type such that it can host the signals and slots necessary for communicating asynchronously with the Data Window, which connects user input to changes in visualization and labeling. It is also the interface for the Settings module, interacting directly with user input and application state. From here the user can add and remove labels, change line and label colors, and so forth. A full list of features is presented below.

It exists as a floating window atop the Data Window, allowing for the user to immediately see the changes they make to the Data Window and its content in the most striking way. This functionality is enabled by Qt's signal and slot system, which is a form of event-driven programming. All of the Qt widgets in the Settings Window are connected to slots in the static Settings class when it is first started.



Settings

Add label

NP | Color... | Delete | ✓ Show label color

Line color | Color...

Show grid | ✓ Show label text | Show durations

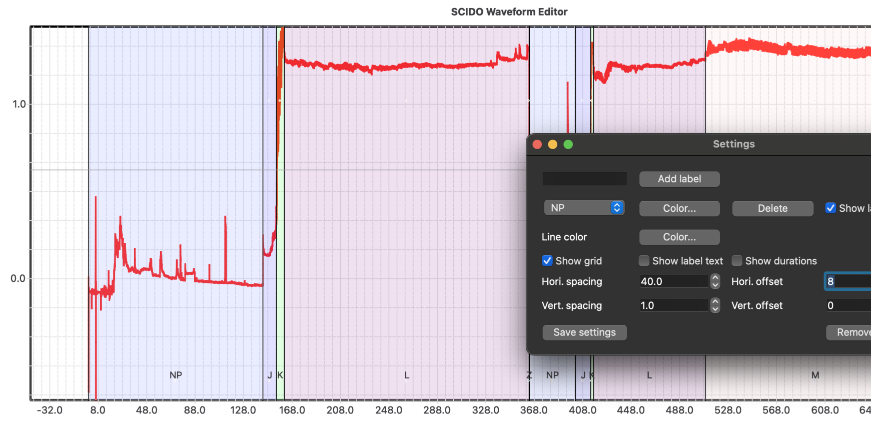
Hori. spacing 100.0 | Hori. offset 0

Vert. spacing 5.0 | Vert. offset 0

Save settings | Remove baseline

Figure 4.4 Settings Window with the color picker for the line color open, and a changed line color

These settings can be saved on request from the user. On Windows, the settings are store to the registry for SCIDO. On MacOS, the settings are storedin the Preferences-folder. It is currently untested on Linux and UNIX systems, and according to the QSettings-documentation it is dependent on the Linux distribution, though generally it comes in the form of local .ini-files.



Settings

Add label

NP | Color... | Delete | ✓ Show l

Line color | Color...

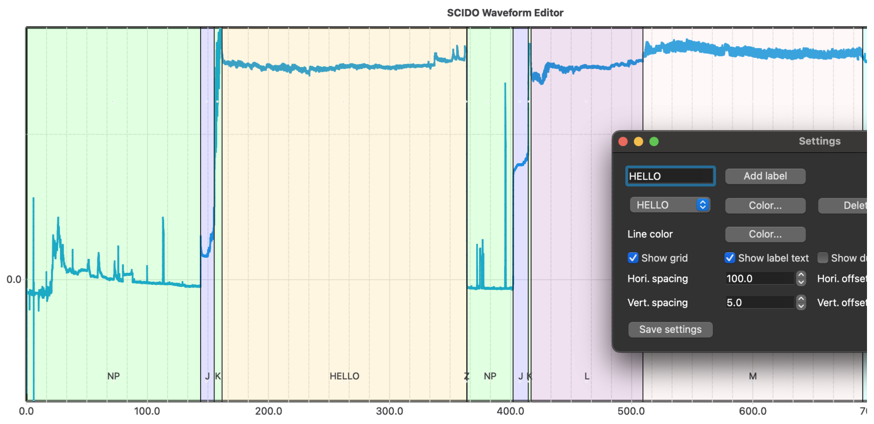
✓ Show grid | Show label text | Show durations

Hori. spacing 40.0 | Hori. offset 8

Vert. spacing 1.0 | Vert. offset 0

Save settings | Remov

Figure 4.5 Settings Window with tight gridline spacing and an offset in the background



Show grid ✓
Show label text ✓
Hori. spacing 100.0
Vert. spacing 5.0

Figure 4.6 New label created with a color set

The following are the parameters that may be adjusted through the Settings Window, and the names in the window:

- Adding and removing labels
- Showing and hiding labels
- Altering the label color (names are changed in the Data Window, by double-clicking)
- Waveform line color
- Showing and hiding the grid
- Horizontal and vertical tick spacing
- Horizontal and vertical tick offset
- Saving the current settings to disk
- Removing baseline (if set)

Warning: too low tick spacing will drastically increase the number of gridlines being drawn, impacting performance.

Chapter 5

Conclusion

5.1 GUI Recommendations

While our primary intention behind building the SCIDO interface was to provide a way for users to interact with our automated labeling solutions, there is great potential to expand SCIDO into a central hub for all EPG labeling processes. Currently, the software supports data loading, automated labeling, manual labeling, and comment viewing. Some possible additions toward a comprehensive software include features for comment creation, automatic data analytics, and live display of incoming data during active experiments and readings.

This live display feature would be the greatest transformation of our software and the largest step toward shaping SCIDO into the primary software used by EPG researchers. To create this functionality is only a matter of allowing the interface to take voltage data inputs from the hardware as well as in the CSV/WINDAQ formats we already support. From here, the software would work as-is, only with an added frequent screen refresh to display updated data as new information is read. This feature exists in the projects of both last year's Auburn-USDA CS Clinic team and this year's Auburn-USDA Engineering Clinic team.

A large additional goal for our liaisons is to expand SCIDO to accommodate automated labeling of plant-feeding insects, including aphids and sharpshooters. Aside from the model training that must be done to accomplish this, there will be some required adjustments to the interface's display functionality. In the software's current state, the complete, multi-hour long voltage data produced by plant-feeding insects will be rendered all at once. This is too much information to track, especially during chart motion, and would slow or even crash the software. To solve this, very long data should either be dynamically loaded in chunks or sub-sampled. Additionally, our methods for determining which transition a user clicked on would benefit from being rewritten to make use of binary search.

The next cohort of developers who will work within the SCIDO code base will be the interns selected to expand the project to support plant-feeding insect data during the summer of 2025. One task that may be beneficial for this team to start with is the implementation of post-processing methods for cleaning the output of our automated labelers, as described below. This functionality leans to the data side of our project, but will, like all features, need to be embedded into the front-end workflow. From here, they will be prepared to expand the software towards the ultimate vision for SCIDO.

5.2 Machine Learning Recommendations

Next steps on the machine learning side of this project include adding post-processing into the GUI, further investigating data augmentation, and adapting these methods other species with different waveform characteristics.

Our post-processing code is currently implemented in `postprocessing.py` and just needs to be integrated into the GUI with a button and some changes to the backend code. We would recommend first integrating the HMM, described in section 3.3.3, which is fully functional as shipped. This would be a good starter project to gain understanding of both the GUI and machine learning sides of the project, in particular how they interact with each other. The HSMM, described in more detail in section 3.3.4, is both the most theoretically promising model and the hardest to implement well (and currently not implemented in a scalable way). This is a good project to undertake next, as it is mostly an algorithms question and requires less knowledge of the existing codebase.

With regards to data augmentation, the methods we tried were fairly rudimentary and we did not have the time to investigate how more sophisticated ones like MixUp as described in Zhang et al. (2017) would have improved performance. Implementing MixUp in particular will require a re-write of how probe data is handled in the program. Currently, we store each probe as a list of labels. MixUp on the other hand will require that we are able to pass around data in the form of a one-hot vector. Additionally, it would be necessary to think about the best way to apply MixUp which assumes data of the same shape to our probes which have a wide variety of shapes.

Finally and most importantly for the next steps of this project, our models were only tested on mosquito waveforms. While this models should be applicable to any waveform classification problem, some will be easier to apply than others. In particular, the training and running time of some of these models scales quadratically with the size of the input recordings. When applying them to other species like sharpshooters which have much longer recordings, it will likely be necessary to figure out a reasonable way of dividing them into shorter segments, like how we divided longer recordings into probes. This can also be a major problem for receptive field reasons, because some of the states can potentially be longer than our receptive field. There are also computational concerns, for example the UNet might run into memory issues. There are potential solutions to workaround this problem, such as gradient checkpointing, but would require some redesign.

5.3 Outcomes

Over the course of this project, we evaluated a variety of machine learning models for EPG labeling along with the SCIDO software for biologists to use them. While there is room for improvement in our models, our use of a UNet and an FCN are groundbreaking on EPG data. Additionally, we are the first to evaluate existing techniques from aphids on mosquitoes. Our SCIDO software is similarly groundbreaking and represents a large leap in EPG labeling from older methods. Through our work on this project we have produced a solid foundation for further development of automated EPG labeling tools.

Appendix A

Installation

A.1 User

The user should be able to select the appropriate executable file for their platform. Currently, the application has been packaged into a single executable file for x86-64 Linux and Windows, and a `.app` application bundle for MacOS Arm. (The application could be built for other platforms if desired.) These are "portable" and do not use an installer — the user should be able to download the file, and run it directly.

A.2 Developer

The GUI is developed using Python and PyQt6. To install the necessary dependencies and run the GUI, use the following sequence of commands:

```
$ cd GUI/
$ python -m venv venv
$ . venv/bin/activate
$ pip install -r requirements.txt
$ python main.py
```

The machine learning code was developed in the `ML/` directory, and only the files needed for the GUI to run were copied into a subdirectory of `GUI/` for packaging. (So, future ML development in this repository should probably take place in `ML/`).

A.3 Packaging

Packaging is done using the using the PyInstaller library (https://pyinstaller.org). (In current state of the repository, packaging is done on the `packaging` branch, currently at commit `9dd55d8`). To build, ensure all dependencies are installed (including PyInstaller), preferably in a virtual environment, and run `build.sh` for Linux/MacOS and `build.ps1` for Windows. These scripts simply call PyInstaller with the appropriate arguments. Note that PyInstaller is only capable of building the application for a certain platform when running on that platform.

Bibliography

Adasme-Carreño, Francisco, Camila Muñoz-Gutiérrez, Josselyn Salinas-Cornejo, and Claudio C. Ramírez. 2015. A2epg: A new software for the analysis of electrical penetration graphs to study plant probing behaviour of hemipteran insects. *Computers and Electronics in Agriculture* 113:128–135. doi:https://doi.org/10.1016/j.compag.2015.02.005. URL https://www.sciencedirect.com/science/article/pii/S0168169915000472.

Akiba, Takuya, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Anonymous. 2007. Orosius orientalis epg. URL https://commons.wikimedia.org/wiki/File:Orosius_orientalis_EPG.jpg. [Online; accessed April 22, 2025].

Bai, Shaojie, J. Zico Kolter, and Vladlen Koltun. 2018. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. URL https://arxiv.org/abs/1803.01271. 1803.01271.

Beltagy, Iz, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. URL https://arxiv.org/abs/2004.05150. 2004.05150.

Cooper, Anastasia M. W., Samuel B. Jameson, Victoria Pickens, Cameron Osborne, Elaine A. Backus, Kristopher Silver, and Dana N. Mitzel. 2024. An electropenetrography waveform library for the probing and ingestion behaviors of culex tarsalis on human hands. *Insect Science* 31(4):1165–1186. doi:https://doi.org/10.1111/1744-7917.13292. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/1744-7917.13292. https://onlinelibrary.wiley.com/doi/pdf/10.1111/1744-7917.13292.

Degirmenci, Alperen. 2014. Introduction to hidden markov models. Harvard University. URL https://scholar.harvard.edu/files/adegirmenci/files/hmm_adegirmenci_2014.pdf. Accessed: 2025-05-01.

Dinh, Quang Dung, Daniel Kunk, Truong Son Hy, Vamsi Nalam, and Phuong D Dao. 2025. Machine learning for automated electrical penetration graph analysis of aphid feeding behavior: Accelerating research on insect-plant interactions. *PLOS ONE* 20(4):1–25. doi:10.1371/journal.pone.0319484. URL https://doi.org/10.1371/journal.pone.0319484.

Ermshaus, Arik, Patrick Schäfer, and Ulf Leser. 2023. Clasp: parameter-free time series segmentation. *Data Mining and Knowledge Discovery* .

Forney, G. David. 1973. The viterbi algorithm. *Proceedings of the IEEE* 61(3):268–278. doi:10.1109/PROC.1973.9030. URL https://ieeexplore.ieee.org/document/1450960.

Gathany, James. 2006. 9258. URL https://phil.cdc.gov/details.aspx?pid=9258. [Online; accessed April 22, 2025].

Hastie, T., R. Tibshirani, and J.H. Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics, Springer. URL https://books.google.com/books?id=eBSgoAEACAAJ.

Long, Jonathan, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. URL https://arxiv.org/abs/1411.4038. 1411.4038.

Murphy, Kevin P. 2002. Hidden semi-markov models (hsmms). URL https://www.cs.ubc.ca/~murphyk/papers/segment.pdf. Accessed: 2025-05-01.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. arXiv:1912.01703.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12:2825–2830.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. URL https://arxiv.org/abs/1505.04597. 1505.04597.

Savitzky, Abraham, and Marcel J. E. Golay. 1964. Smoothing and differentiation of data by simplified least squares procedures. *Analytical Chemistry* 36(8):1627–1639. doi:10.1021/ac60214a047. URL https://doi.org/10.1021/ac60214a047.

Truong, Charles, Laurent Oudre, and Nicolas Vayatis. 2020. Selective review of offline change point detection methods. *Signal Processing* 167:107,299. doi:https://doi.org/10.1016/j.sigpro.2019.107299. URL https://www.sciencedirect.com/science/article/pii/S0165168419303494.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention is all you need. URL https://arxiv.org/abs/1706.03762. 1706.03762.

Virtanen, Pauli, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17:261–272. doi:10.1038/s41592-019-0686-2.

Willett, Denis S., Justin George, Nora S. Willett, Lukasz L. Stelinski, and Stephen L. Lapointe. 2016. Machine learning for characterization of insect vector feeding. *PLOS Computational Biology* 12(11):e1005,158. doi:10.1371/journal.pcbi.1005158. URL https://app.dimensions.ai/details/publication/pub.1014451934.

Zhang, Hongyi, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. 2017. mixup: Beyond empirical risk minimization. arXiv:1710.09412.

Zheng, Sixiao, Jiachen Lu, Hengshuang Zhao, Xiatian Zhu, Zekun Luo, Yabiao Wang, Yanwei Fu, Jianfeng Feng, Tao Xiang, Philip H. S. Torr, and Li Zhang. 2020. Rethinking semantic segmentation from a sequence-to-sequence perspective with transformers. arXiv:2012.15840.