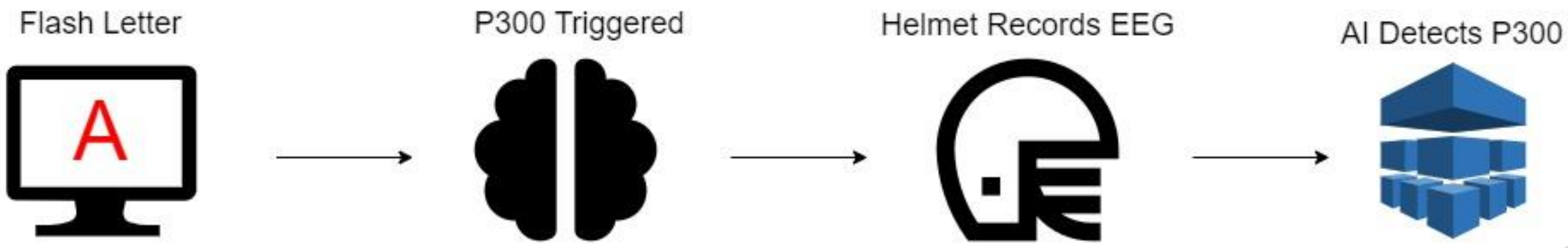
Brain-Computer Interfaces

Outline

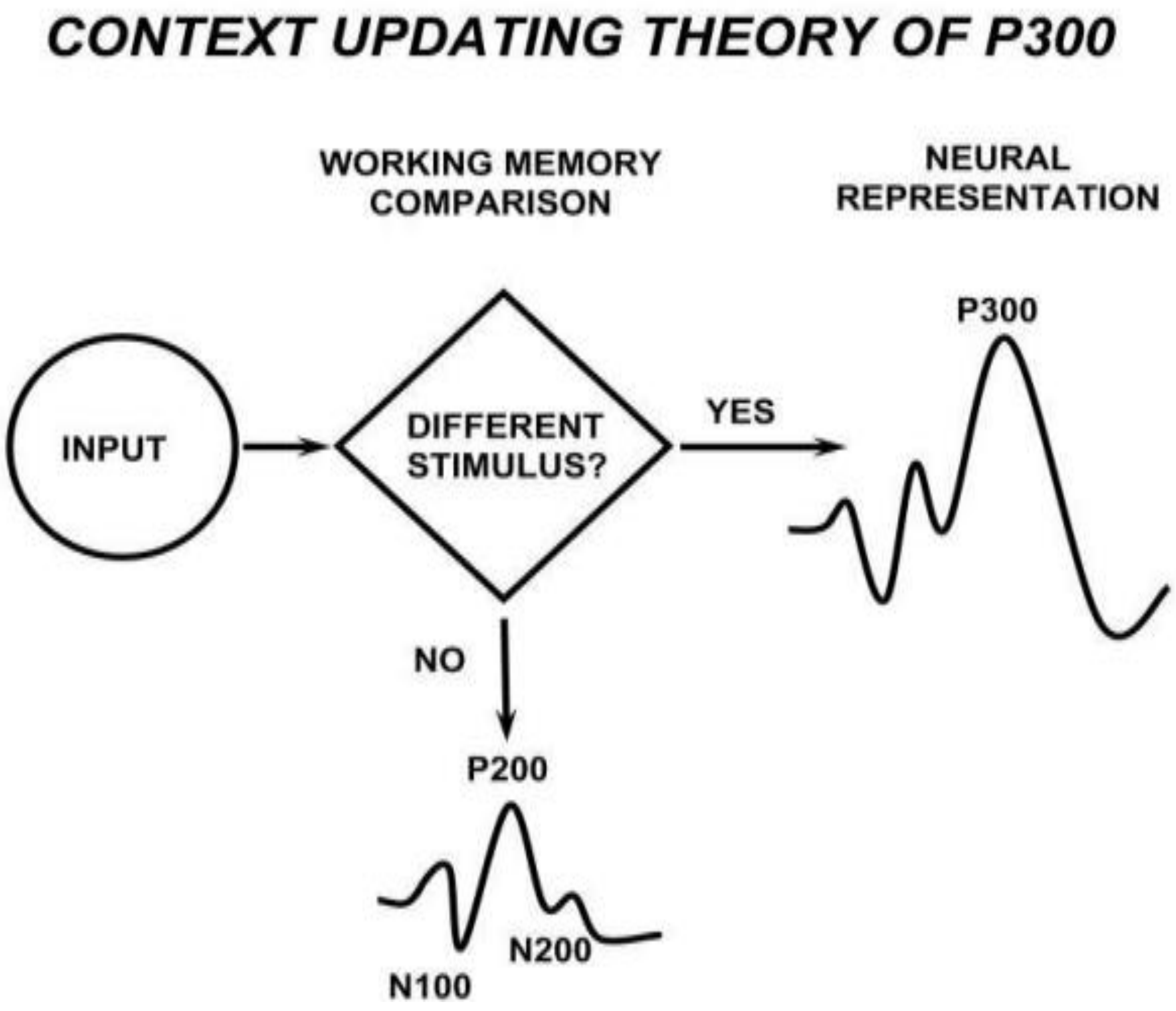
- Project Goals
- P300 Overview
- Speller Interface
- OpenBCI Hardware and API
- EEG Dataset
- Keras Neural Network (CUDA)
- User Interface
- Transfer Learning and Data Collection
- Challenges
- Lessons Learned
- Demo

Project Goals

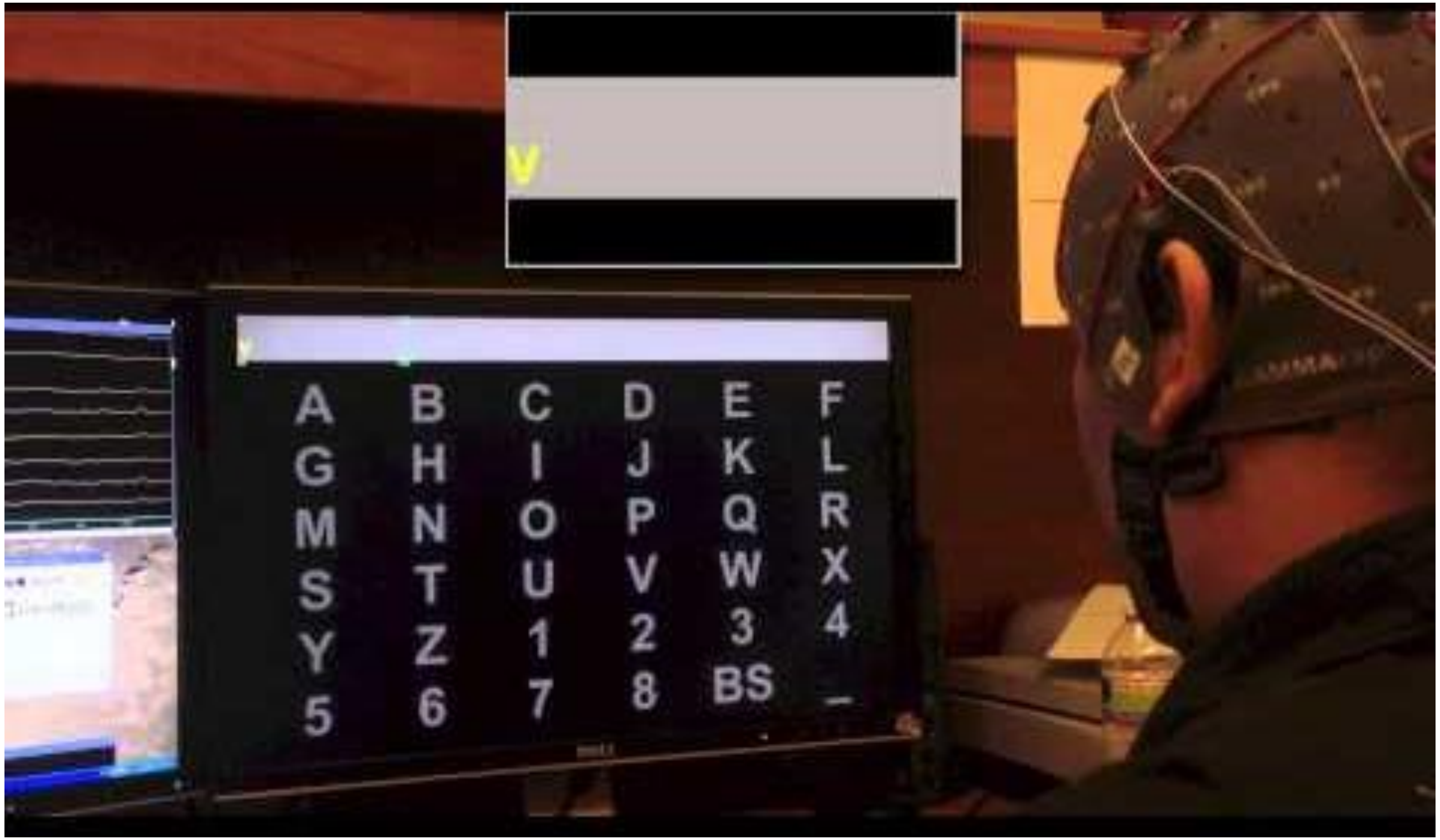
- Learn about Electroencephalography (EEG)
- Learn what BCI is and how (high level) a P300 BCI speller works
- Implement basic neural networks and machine learning
- Use hardware involved in a dry electrode EEG cap/helmet
- Understand how to approach creating a basic P300 BCI speller



P300 Overview

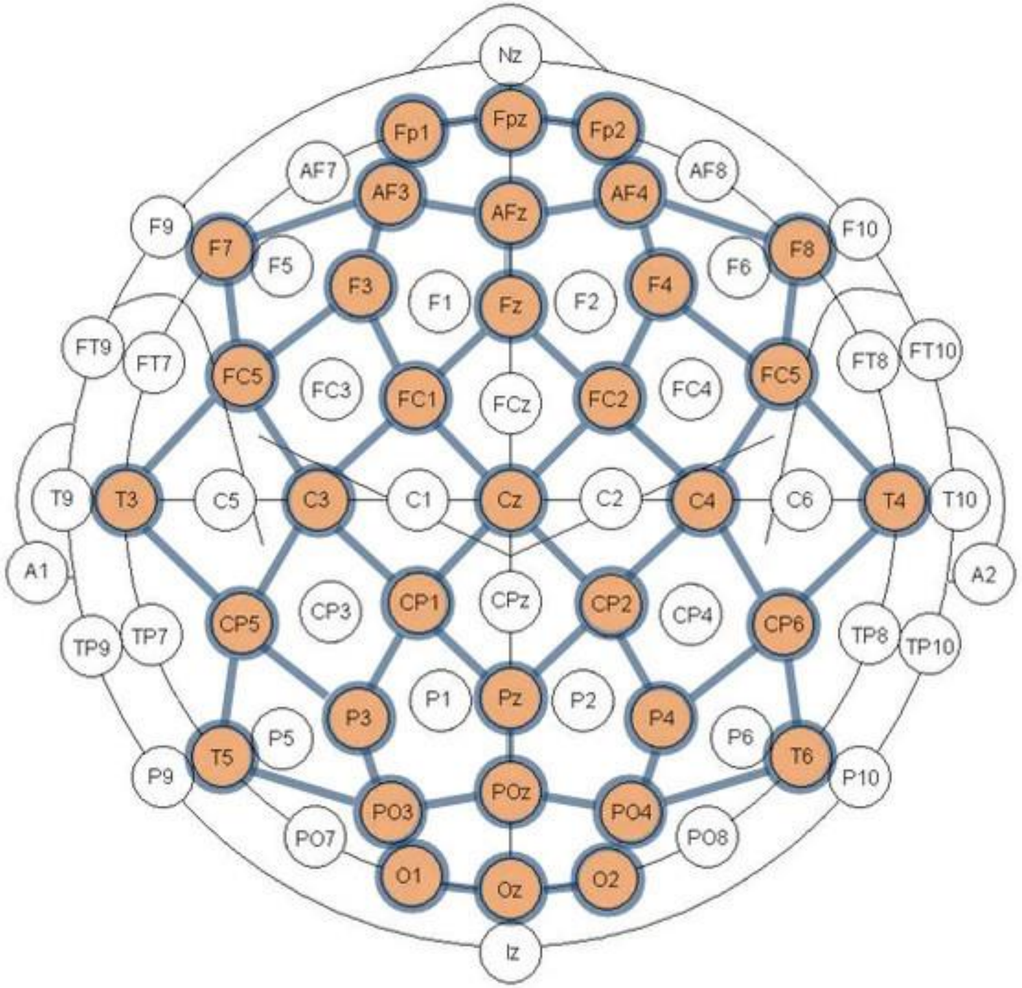


Speller Interface



OpenBCI Hardware and API

Ultracortex Mark IV
Node Locations (35 total)

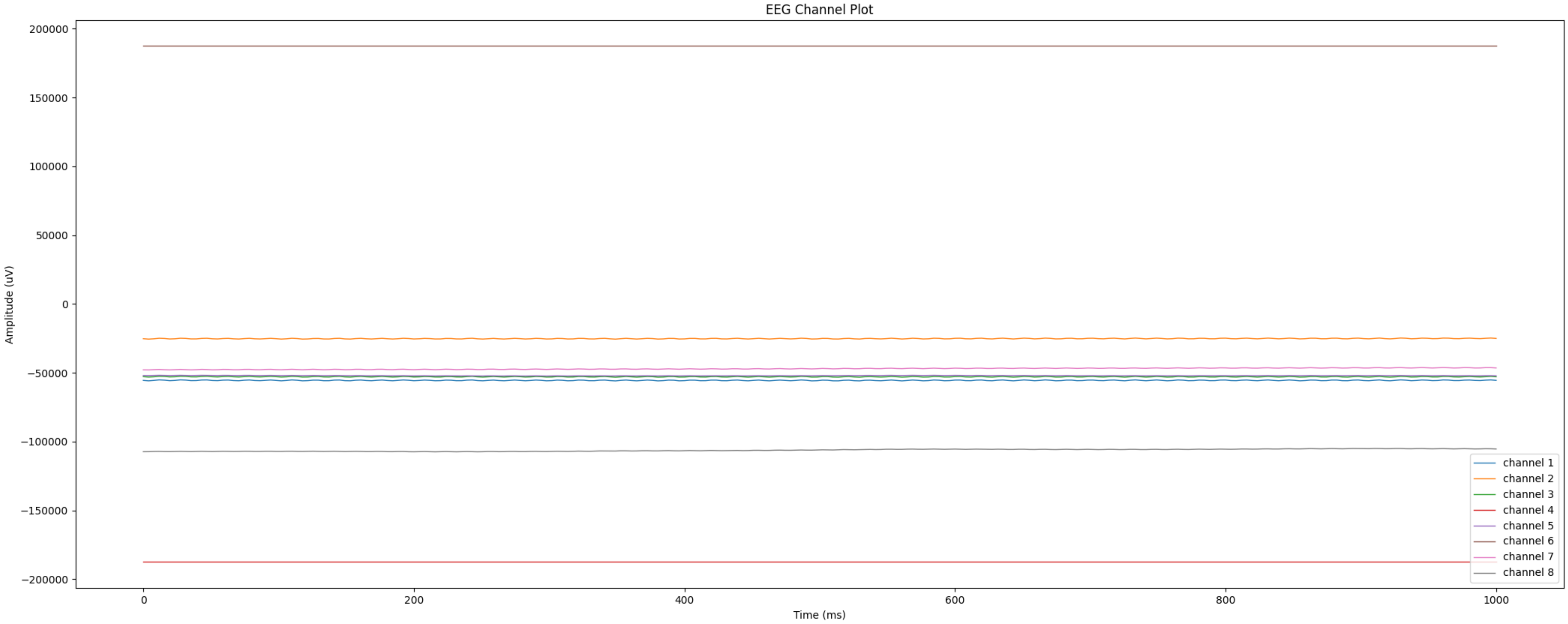


Based on the internationally accepted **10-20 System** for electrode placement in the context of EEG research

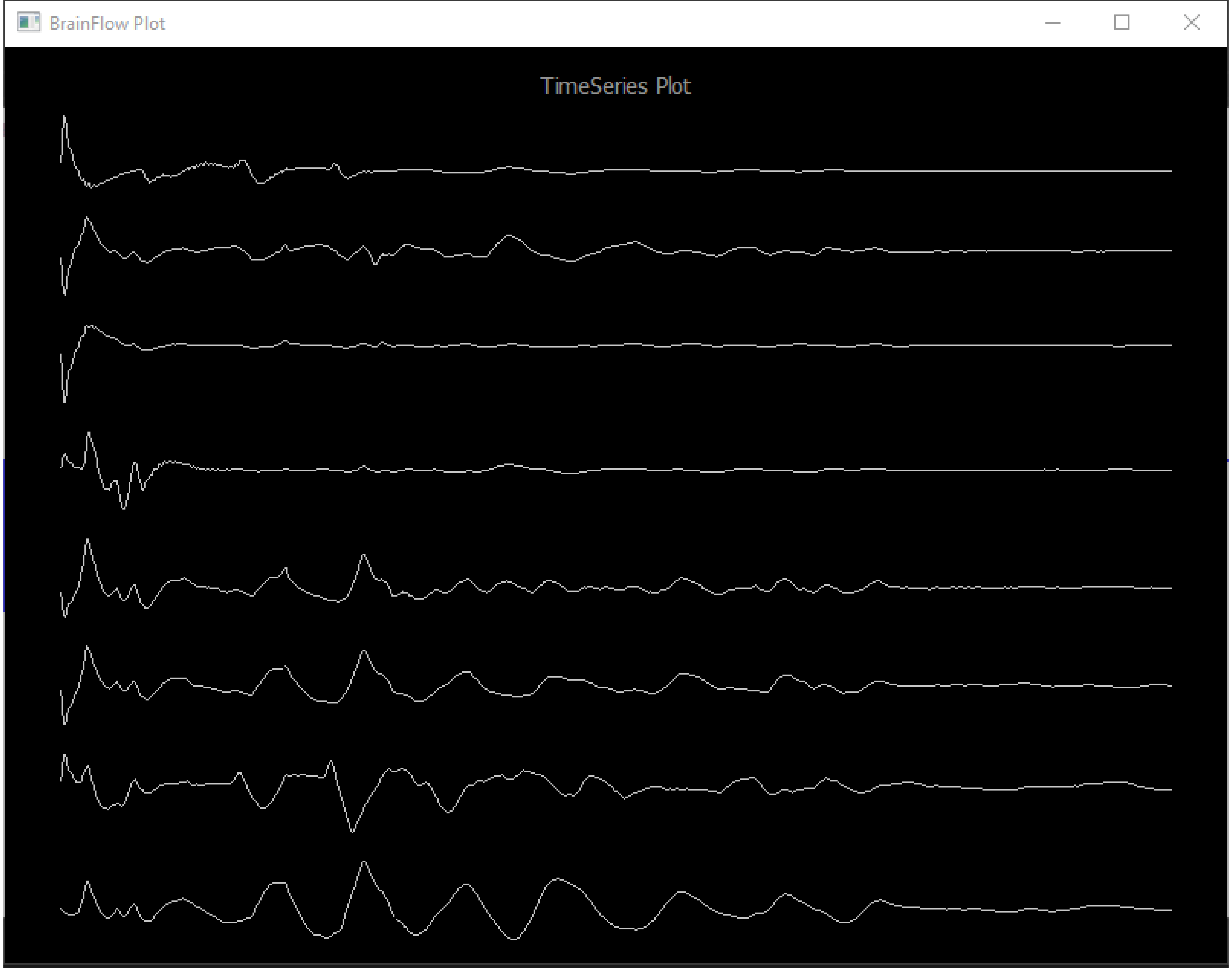
8 Channels:
Fpz
C2
C3
C4
P2
P3
P4
O1
O2

Signal Filtering

```
# apply filters
for i in range(len(channels)):
    # Subtract trend from data (simply removes long-term increase or decrease in the level of the time series.)
    DataFilter.detrend(channels[i], DetrendOperations.CONSTANT.value)
    # Perform band pass filter in-place between 3Hz and 45Hz using 2nd order filter
    DataFilter.perform_bandpass(channels[i], sampling_rate, 3.0, 45.0, 2, FilterTypes.BUTTERWORTH.value, 0)
    # Perform band stop (i.e. notch) filter (inverse of bandpass) between 48Hz and 52Hz using 2nd order filter
    DataFilter.perform_bandstop(channels[i], sampling_rate, 48.0, 52.0, 2, FilterTypes.BUTTERWORTH.value, 0)
    # Perform another band stop filter between 58Hz and 62Hz using 2nd order filter
    DataFilter.perform_bandstop(channels[i], sampling_rate, 58.0, 62.0, 2, FilterTypes.BUTTERWORTH.value, 0)
return channels
```



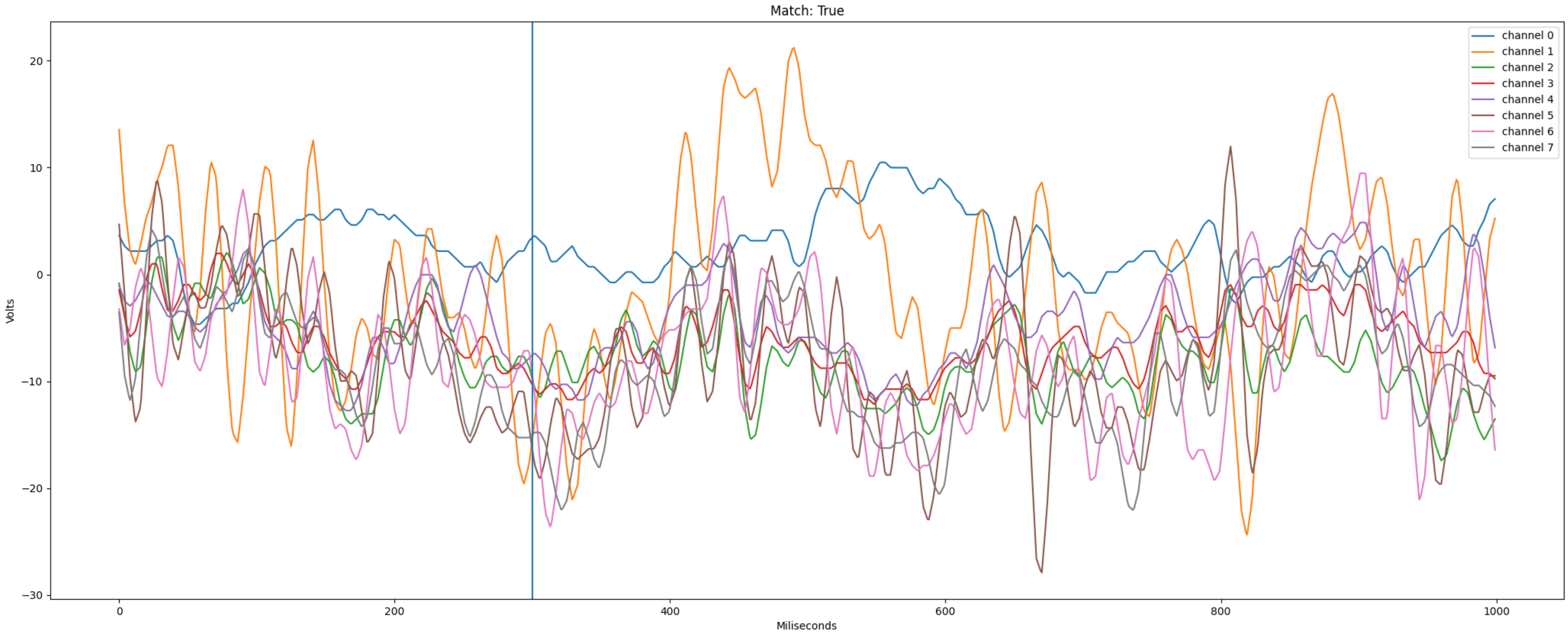
EEG Channel Plot
200000
150000
100000
50000
0
−50000
−100000
−150000
−200000
Amplitude (uV)
0
200
400
600
800
1000
Time (ms)
channel 1
channel 2
channel 3
channel 4
channel 5
channel 6
channel 7
channel 8


BrainFlow Plot
TimeSeries Plot

EEG Dataset

```
# co2a0000364.rd
# 120 trials, 64 chans, 416 samples 368 post_stim samples
# 3.906000 msecs uV
# S1 obj , trial 0
# FP1 chan 0
0 FP1 0 -8.921
0 FP1 1 -8.433
0 FP1 2 -2.574
0 FP1 3 5.239
...
0 FP1 253 4.262
0 FP1 254 5.727
0 FP1 255 8.169
# FP2 chan 1
0 FP2 0 0.834
0 FP2 1 3.276
0 FP2 2 5.717
```

Data Shape:
3,725 Trials
8 Channels/Trial
256 Samples/Channel
Roughly 7.5 million datapoints

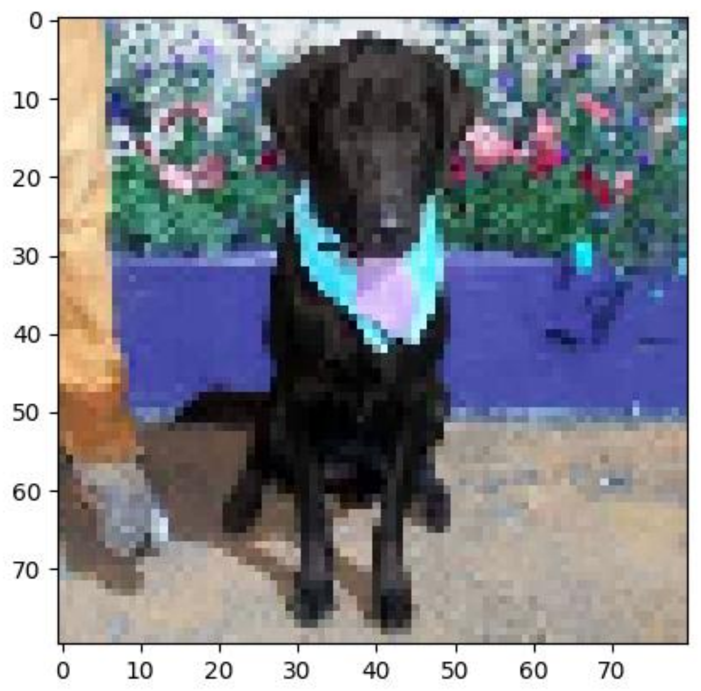

Match: True
channel 0
channel 1
channel 2
channel 3
channel 4
channel 5
channel 6
channel 7
20
10
0
−10
−20
−30
Volts
0
200
400
600
800
1000
Miliseconds

Feedforward Neural Network Practice

```
#define model 784 -> 64 -> 10 nodes
model = nn.Sequential(
    nn.Linear(in_features=28*28, out_features=64, bias=True),
    nn.ReLU(),
    nn.Linear(in_features=64, out_features=64, bias=True),
    nn.ReLU(),
    nn.Linear(in_features=64, out_features=10, bias=True)
)
```

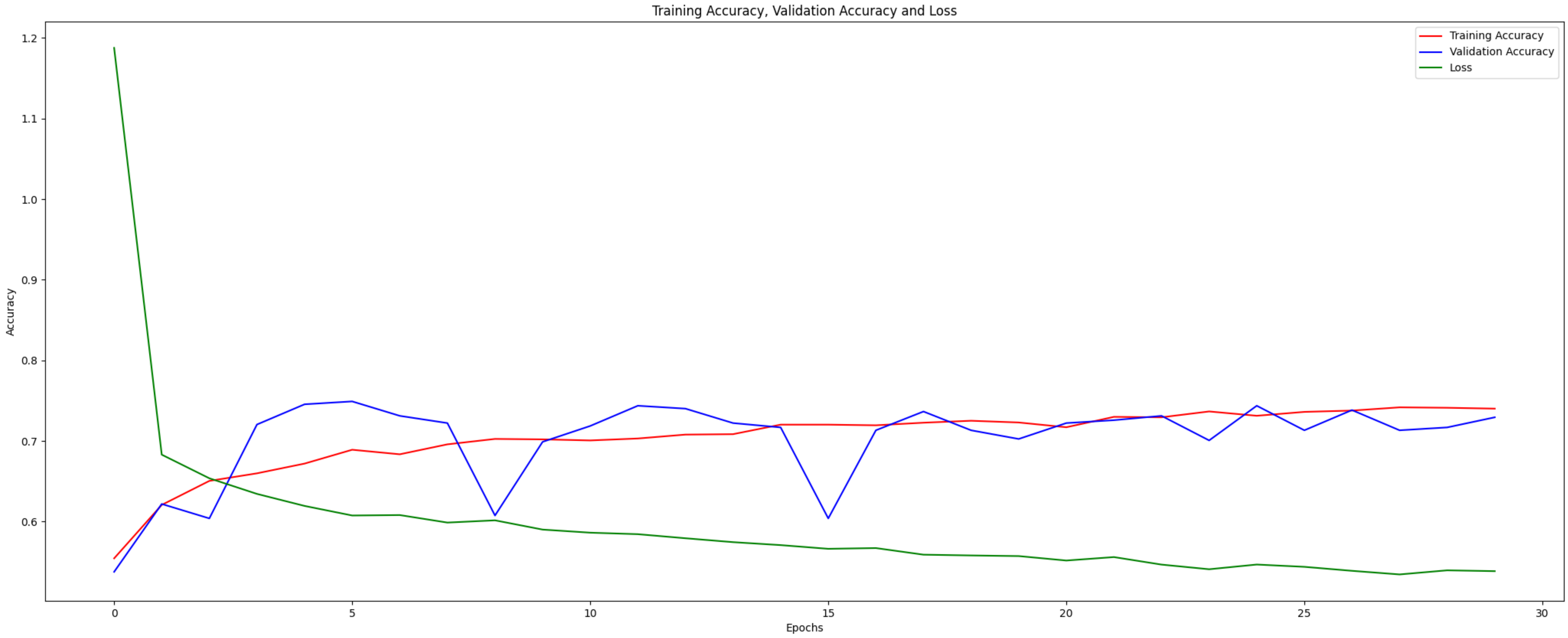
Convolutional Neural Network Practice

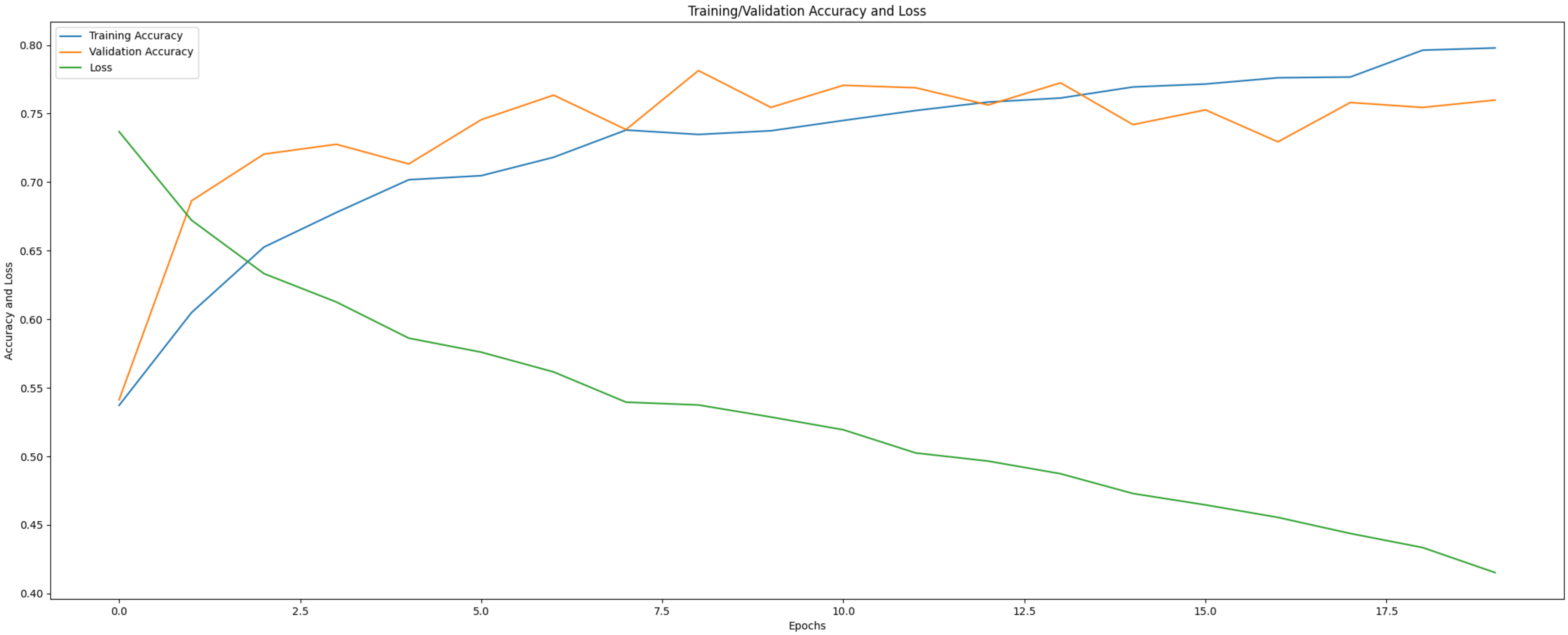
```
# BUILD NETWORK
network = models.Sequential()
# conv layer 1
# 80, 80, 3 corresponds to image shape
network.add(layers.Conv2D(128,(3,3),input_shape=(80,80,3)))
network.add(layers.LeakyReLU())
network.add(layers.MaxPooling2D(pool_size=(2,2)))
# conv layer 2
network.add(layers.Conv2D(80,(3,3)))
network.add(layers.LeakyReLU())
network.add(layers.MaxPooling2D(pool_size=(2,2)))
# conv layer 3
network.add(layers.Conv2D(32,(3,3)))
network.add(layers.LeakyReLU())
network.add(layers.MaxPooling2D(pool_size=(2,2)))
# flatten layers
network.add(layers.Flatten())
# intermediate layer
network.add(layers.Dense(128,  kernel_regularizer=regularizers.l2(0.001)))
network.add(layers.LeakyReLU())
# final layer
# final layer has 1 node, either 0 or 1 for cat or dog
network.add(layers.Dense(1,activation='sigmoid'))
# DEFINE OPTIMIZER AND LOSS FUNCTION
network.compile(optimizer='rmsprop',loss='binary_crossentropy',metrics=['accuracy'])
```



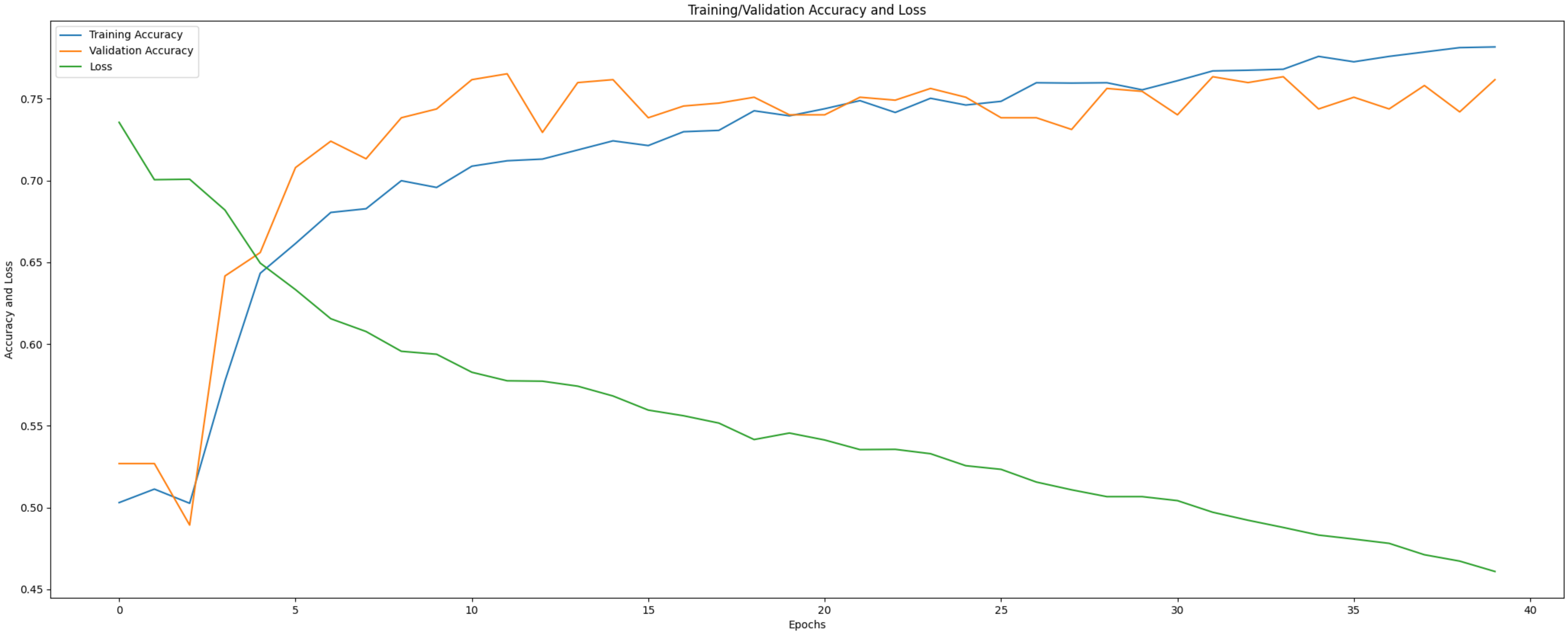
Keras Neural Network (CUDA)

```
# BUILD NETWORK
network = models.Sequential()
# Convolutional layers
network.add(layers.Conv1D(filters=128, kernel_size=3, padding='same', input_shape=(8, 256), data_format='channels_first'))
network.add(layers.Conv1D(filters=64, kernel_size=7, padding='same'))
network.add(layers.Conv1D(filters=32, kernel_size=11, padding='same'))
network.add(layers.MaxPool1D(pool_size=2))
network.add(layers.Flatten())
# Dense layer 1, normalized
network.add(layers.Dense(units=1024))
network.add(layers.LayerNormalization())
network.add(layers.ELU())
# Dense layer 2
network.add(layers.Dense(units=512))
# Dense layer 3
network.add(layers.Dense(units=256))
# Dense layer 4, normalized
network.add(layers.Dense(units=128))
network.add(layers.LayerNormalization())
network.add(layers.ELU())
# Final layer, binary classifier
network.add(layers.Dense(units=1, activation='sigmoid'))
```

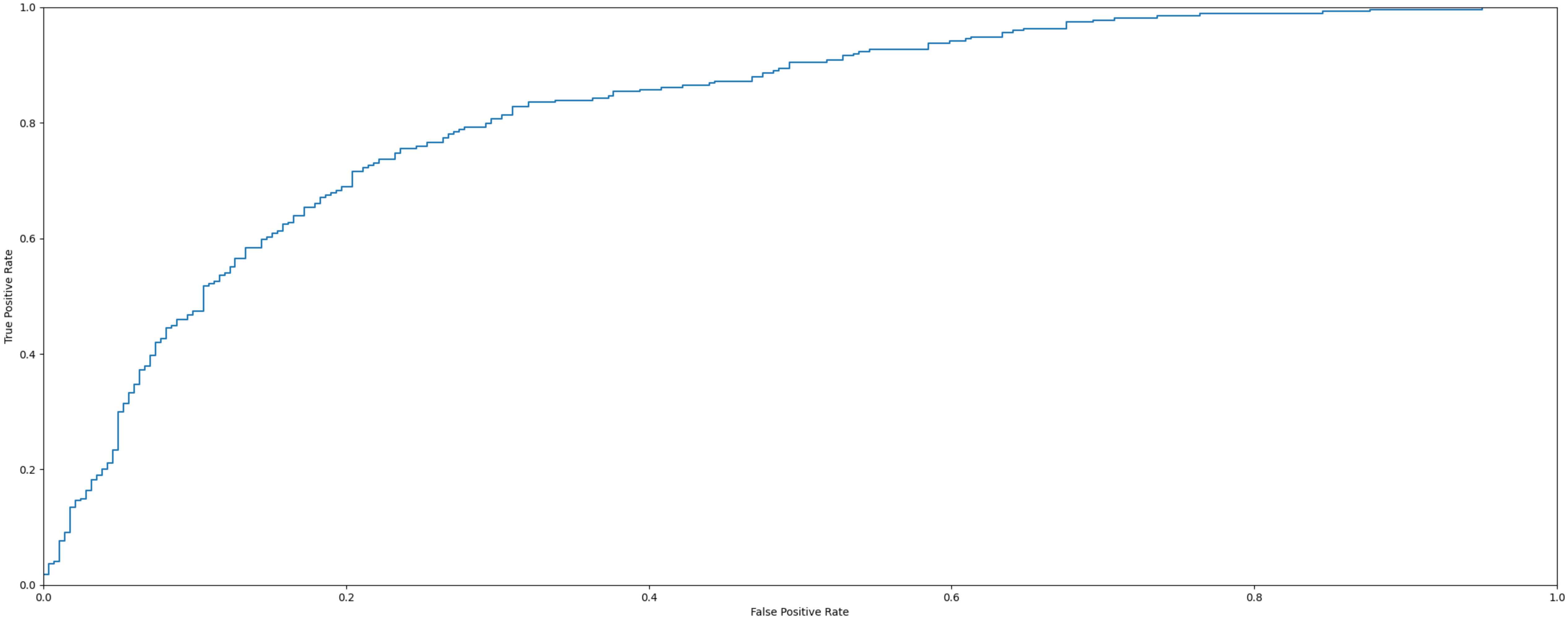

Training Accuracy, Validation Accuracy and Loss
Training Accuracy
Validation Accuracy
Loss
Accuracy
Epochs
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0
5
10
15
20
25
30



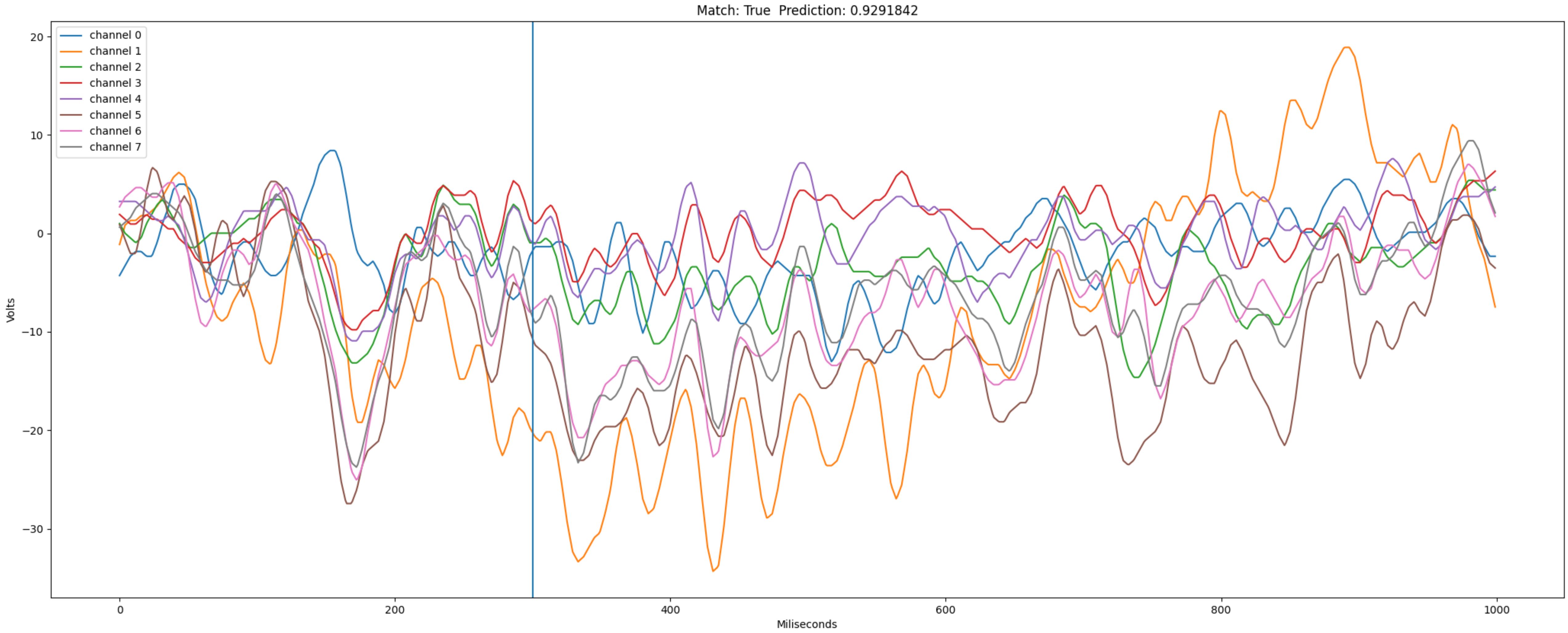
Training/Validation Accuracy and Loss
Training Accuracy
Validation Accuracy
Loss
Accuracy and Loss
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

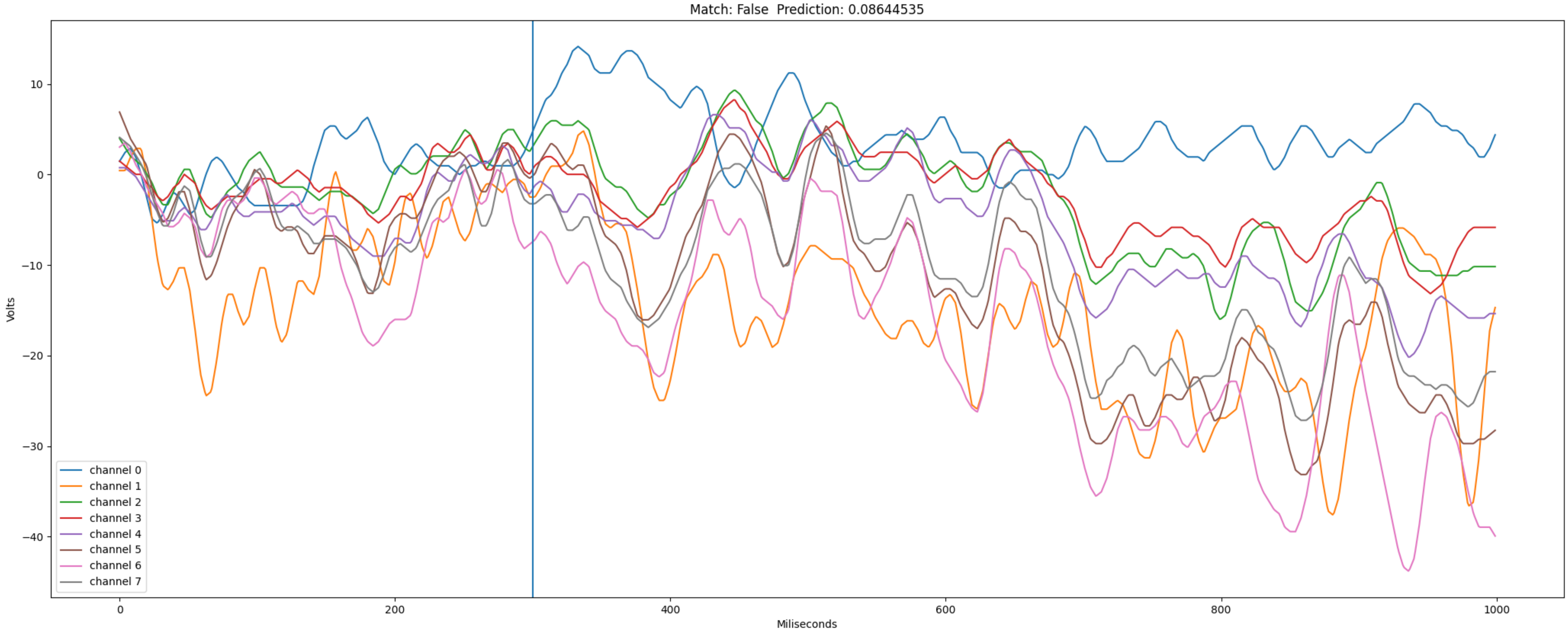


Training/Validation Accuracy and Loss
Training Accuracy
Validation Accuracy
Loss
Accuracy and Loss
0.45
0.50
0.55
0.60
0.65
0.70
0.75
0
5
10
15
20
25
30
35
40
Epochs

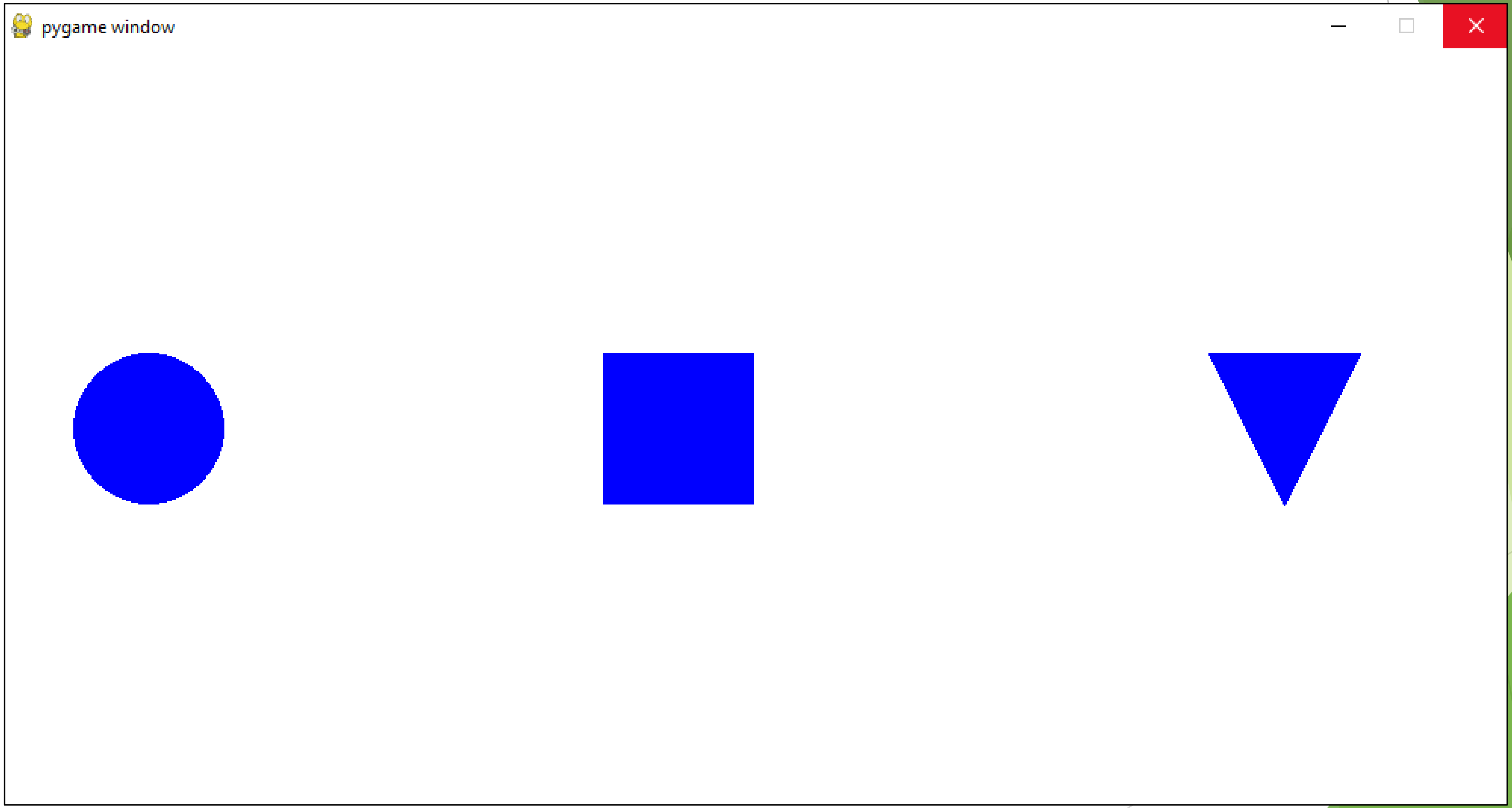


True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0


Match: True Prediction: 0.9291842
channel 0
channel 1
channel 2
channel 3
channel 4
channel 5
channel 6
channel 7
Volts
Miliseconds
20
10
0
−10
−20
−30
0
200
400
600
800
1000

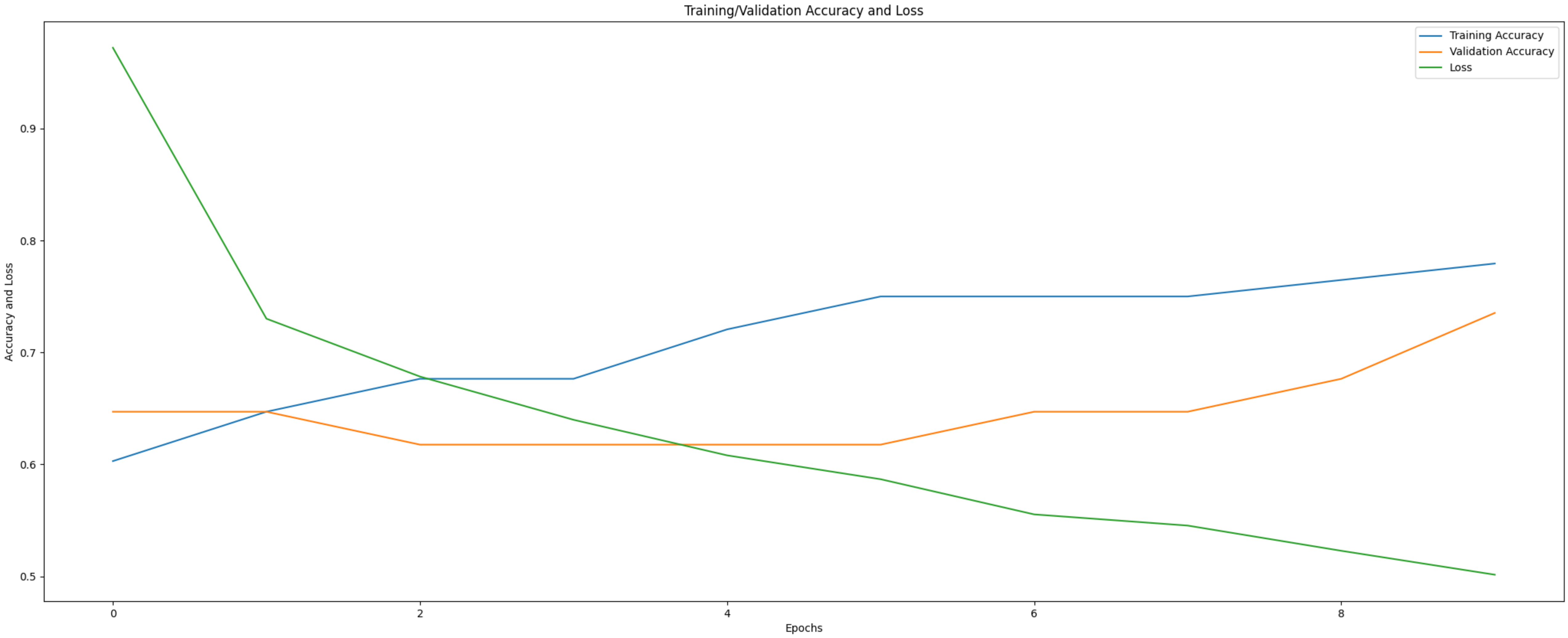

Match: False Prediction: 0.08644535
Volts
Miliseconds
10
0
−10
−20
−30
−40
0
200
400
600
800
1000
channel 0
channel 1
channel 2
channel 3
channel 4
channel 5
channel 6
channel 7

User Interface and Transfer Learning

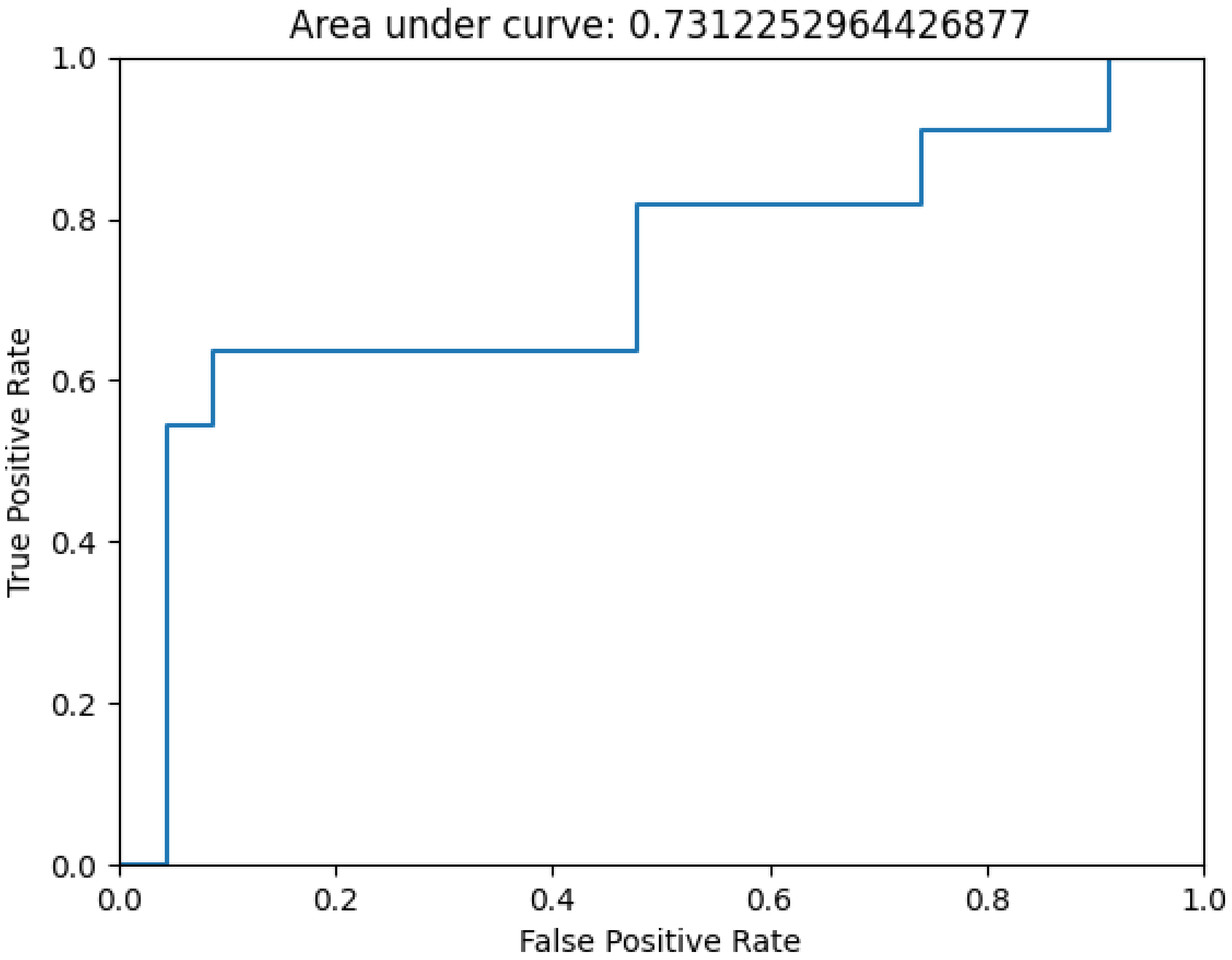


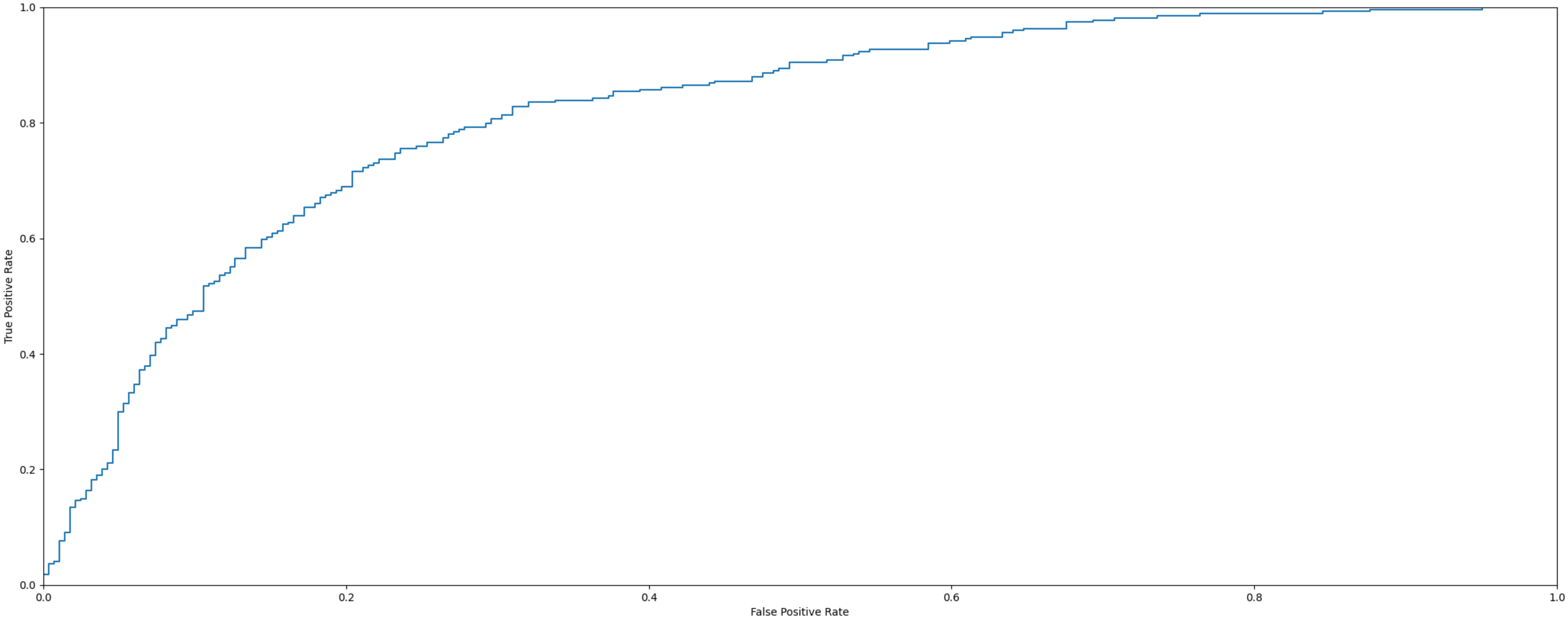
Multithreading

```python
# we will use a thread lock to prevent potential race conditions when updating lists
_lock = threading.Lock()

# function to be called by threads
def prediction_thread(shape, board, channel_nums, sampling_rate, network, prediction_list, averages):
    # sleep 1 second so that get_data() gets the correct data
    time.sleep(1)
    channel_data = cyton_funcs.get_data(board, channel_nums, sampling_rate)

    # pass data to nueral network
    prediction_array = network.predict(np.array([channel_data,]), verbose=0)
    prediction = prediction_array[0][0]

    # add shape and prediction to list, basically a history of each prediction
    with _lock: # lock this segment of code so that only one thread at a time may enter
        prediction_list.append([shape, prediction])
        # average the predicton value for given shape
        if shape == 'Circle':
            averages[0] = (averages[0] + prediction) / averages[3]
            averages[3] += 1
        if shape == 'Square':
            averages[1] = (averages[1] + prediction) / averages[4]
            averages[4] += 1
        if shape == 'Triangle':
            averages[2] = (averages[2] + prediction) / averages[5]
            averages[5] += 1
```



Training/Validation Accuracy and Loss
Training Accuracy
Validation Accuracy
Loss
Accuracy and Loss
0.5
0.6
0.7
0.8
0.9
0
2
4
6
8
Epochs


Area under curve: 0.7312252964426877
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0



1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate

Challenges

- Small initial training dataset
- Signal filtering for live data
- Neural network implementation
- Time required for transfer learning data collection
- Development on different OS (graphics libraries)
- Safe data structures for multithreaded applications
- Time offsets for data collection
- Keras library limitations

Lessons Learned

- Size of datasets required for AI
- Importance of signal processing and data filtering
- Strategies for managing large coding projects
- Hardware and real-world limitations
- Managing expectations for research

Demo