Introduction to Neural Screenomics

December 9, 2024

1 Research background

Name: Rinseo Park

Last Updated: December 1, 2024

Notebook URL:

https://colab.research.google.com/drive/1fd9oKozft0i-lLF8ElpULkC7yc7FTsjS?usp=sharing

---

During the last few years, *the Human Screenome Project* (Reeves et al., 2021) has amassed a large repository of screenshot sequences – detailed records obtained every five seconds for many weeks/months of what appeared on hundreds of study participants' smartphone screens. These image time-series data constitute a large library of humans' action possibilities – detailed observations of how individuals engage in a diverse array of complex tasks, how they switch among tasks, and how they simultaneously manage and mismanage multiple competing motivations. The data serve as an ideal platform for discovering models that describe how humans' navigate through a multimodal digital environment while balancing multiple competing goals and managing multiple interruptions and distractions.

Drawing parallel with multi-channel neural recordings that indicate how neural activity manifests across different brain regions, trajectories of digital media use may indicate how a variety of motivational states quickly change, evolve, and eventually turn into an action. Thus, this project uses spike sorting techniques in neuroscience to identify individuals' dynamic motivations.

2 Environment setup

This is a PyTorch (version 2.0) implementation of the Neural Screenomics Project.

[ ]:
```
# Import PyTorch modules
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.distributions as dist
import numpy as np

# We'll use a few functions from scipy
from scipy.signal import find_peaks
```

```
from scipy.optimize import linear_sum_assignment

# Plotting stuff
import matplotlib.pyplot as plt
from matplotlib.cm import get_cmap
from matplotlib.gridspec import GridSpec
import seaborn as sns

# Some helper utilities
from tqdm.auto import trange
```

[ ]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
if not torch.cuda.is_available():
    print('WARNING: THIS LAB IS BEST PERFORMED ON A MACHINE WITH A GPU!')
    print('WARNING: IF USING COLAB, PLEASE SELECT A GPU RUNTIME!')
```

```
WARNING: THIS LAB IS BEST PERFORMED ON A MACHINE WITH A GPU!
WARNING: IF USING COLAB, PLEASE SELECT A GPU RUNTIME!
```

[ ]:
```
# Initialize a color palette for plotting
# Please visit the link above for more details!
```

```
Showcasing the palette. Each color corresponds to a neuron.
```

3 Model implementation

Recent innovations in technology are opening possibility to observe action potentials (spikes) of single neurons from recordings of the local electrical activity around tiny high-density electrodes lowered into the brain State-of-the-art spike sorting algorithms in neuroscience, such as KiloSort (Pachitariu et al., 2016), can identify the dynamic overlapping activity of hundreds of specific neurons at millisecond timescales. This section contains a code implementation of spike sorting based on **Convolutional Nonnegative Matrix Factorization (Conv-NMF)** algorithm.

3.1 Conv-NMF algorithm

[ ]:
```
def log_likelihood(templates, amplitudes, data, noise_std):
    """Evaluate the log likelihood"""
    K, N, D = templates.shape
    _, T = data.shape
```

```
    # Compute the model prediction by convolving the amplitude and templates
    pred = F.conv1d(amplitudes, templates.flip(dims=(-1,)).permute(1,0,2),
                    padding=D-1)
    pred = pred[:,:-(D-1)]

    # Evaluate the log probability using dist.Normal
    ll = dist.Normal(pred,noise_std).log_prob(data).sum()

    # Return the log probability normalized by the data size
    return (ll / (N * T)).to('cpu')

def update_residual(neuron, footprints, profiles, amplitudes, residual):
    """Add the specified neurons' template convolved with its amplitude
    to the residual.
    """
    K, N, R = footprints.shape
    _, _, D = profiles.shape
    _, T = residual.shape

    # Convolve this neuron's amplitude with its weighted temporal factor.
    # Project the result using the neuron's channel factor.
    # Add that result to the residual.
    conv = F.conv1d(amplitudes[neuron].unsqueeze(0), profiles.flip(dims=(-1,)),
                    padding=D-1)
    conv = conv[neuron,:-(D-1)].unsqueeze(0)
    residual += torch.matmul(footprints[neuron],conv)

def downdate_residual(neuron, footprints, profiles, amplitudes, residual):
    """Subtract the specified neurons' template convolved with its amplitude
    to the residual.
    """
    K, N, R = footprints.shape
    _, _, D = profiles.shape
    _, T = residual.shape

    # Convolve this neuron's amplitude with its weighted temporal factor.
    # Project the result using the neuron's channel factor.
    # Subtract that result to the residual.
    conv = F.conv1d(amplitudes[neuron].unsqueeze(0),
                    profiles[neuron].flip(dims=(-1,)).unsqueeze(0), padding=D-1)
    conv = conv[:,:-(D-1)]
    residual -= torch.matmul(footprints[neuron],conv)

def _update_amplitude_fast(neuron, footprints, profiles, amplitudes, residual,
                           noise_std=1.0, amp_rate=5.0):
    K, N, R = footprints.shape
    _, _, D = profiles.shape
```

```
    _, T = residual.shape

    new_amplitudes = torch.zeros(T, device=device)

    # Compute the score by projecting the residual (U_k^T R_k)
    # and cross-correlating with the weighted temporal factor (V_k^T)

    # project the residual onto the channel factor for this neuron
    proj_residual = torch.matmul(footprints[neuron].T,residual)

    # correlate the projected residual with the weighted temporal factor
    score = F.conv1d(proj_residual,
                     profiles[neuron].unsqueeze(0), padding=D-1).to("cpu")
    score = score[:,(D-1):].squeeze()

    # Find the peaks in the cross-correlation
    peaks, props = find_peaks(score,
                              height=(noise_std**2)*amp_rate,
                              distance=D)
    heights = torch.tensor(props['peak_heights'],
                           dtype=torch.float32, device=device)

    # Update the amplitudes for this neuron in place
    new_amplitudes[peaks] = heights

    return new_amplitudes

def _update_template_factors(neuron, footprints, profiles,
                             amplitudes, residual, template_rank=1):
    N = footprints.shape[1]
    D = profiles.shape[2]
    T = residual.shape[1]

    new_footprint = torch.zeros((N, template_rank))
    new_profile = torch.zeros((template_rank, D))

    # Check if the factor is used. if not, generate a random target.
    if amplitudes[neuron].sum() < 1:
        target = generate_templates(N, D, 1)[0]
    else:
        # Compute the target (inner product of residual and regressors)
        delay = torch.eye(D, device=device)
        regressors = F.conv1d(amplitudes[neuron].reshape(1, 1, T),
                              delay.unsqueeze(1).flip(dims=(2,)),
                              padding=D-1)[0, :, :-(D-1)]
        target = residual @ regressors.T
```

```
        pass

    # Project the target onto the set of normalized rank-K templates
    # and keep only the spatial factors (U_n) and the weighted temporal
    # factors (V_n^T)
    U, S, V = torch.svd(target)

    # Truncate the SVD and normalize the singular values
    S = S[..., :template_rank]

    # Truncate, weight, and transpose the factors as appropriate
    new_footprint = U[..., :template_rank]
    new_profile = torch.matmul(torch.diag(S) / torch.norm(S),
                               V[..., :template_rank].T)

    return new_footprint.to('cpu'), new_profile.to('cpu')

def map_estimate_fast(templates,
                      amplitudes,
                      data,
                      num_iters=20,
                      template_rank=1,
                      noise_std=1.0,
                      amp_rate=5.0,
                      tol=1e-06):
    """Fit the templates and amplitudes by maximum a posteriori (MAP) estimation
    """
    K, N, D = templates.shape

    # Make a fancy reusable progress bar for the inner loops over neurons.
    outer_pbar = trange(num_iters)
    inner_pbar = trange(K)
    inner_pbar.set_description("updating neurons")

    # Initialize the residual
    residual = data - F.conv1d(amplitudes.unsqueeze(0),
                               templates.permute(1, 0, 2).flip(dims=(2,)),
                               padding=D-1)[0, :, :-(D-1)]

    # Initialize the template factors
    U, S, V = torch.svd(templates)
    U = U[..., :template_rank]
    S = S[..., :template_rank]
    V = V[..., :template_rank]
    footprints = U
    profiles = V * S.unsqueeze(1)
    profiles = profiles.permute(0, 2, 1)
```

```
    # Track log likelihoods over iterations
    lls = [log_likelihood(templates, amplitudes, data, noise_std=noise_std)]

    # Coordinate ascent
    for itr in outer_pbar:

        # Update neurons one at a time
        inner_pbar.reset()
        for k in range(K):
            # Update the residual in place (add a_k \circledast W_k)
            update_residual(k, footprints, profiles, amplitudes, residual)

            # Update the template and amplitude with the residual
            amplitudes[k] = _update_amplitude_fast(
                k, footprints, profiles, amplitudes, residual,
                noise_std=noise_std, amp_rate=amp_rate)

            footprints[k], profiles[k] = _update_template_factors(
                k, footprints, profiles, amplitudes, residual,
                template_rank=template_rank)

            # Downdate the residual in place (subtract a_k \circledast W_k)
            downdate_residual(k, footprints, profiles, amplitudes, residual)

            # Step the progress bar
            inner_pbar.update()

        # Reconstruct the templates in place
        templates[:] = footprints @ profiles

        # Compute the log likelihood
        lls.append(log_likelihood(templates, amplitudes, data,
                                  noise_std=noise_std))

        # Check for convergence
        if abs(lls[-1] - lls[-2]) < tol:
            print("Convergence detected!")
            break

    return torch.stack(lls)

def predict_data(templates, amplitudes):
  _, _, D = templates.shape

  # Compute the target (inner product of amplitudes and templates)
  data_pred = F.conv1d(amplitudes.unsqueeze(0),
```

```
                        templates.permute(1, 0, 2).flip(dims=(2,)),
                        padding=D-1)[0, :, :-(D-1)]
  return(data_pred)

def calculate_r2(data, templates, amplitudes):

  # Compute the target (inner product of amplitudes and templates)
  data_pred = predict_data(templates, amplitudes)

  # Compute the variance explained (SSR) / total variance (SST)
  SSR = ((data - data_pred) ** 2).sum()
  SST = ((data - data.mean()) ** 2).sum()
  R2 = 1 - SSR/SST
  return(R2)
```

3.2 Parameter initialization

[ ]:
```
def generate_templates(num_channels, len_waveform, num_neurons):
    # Make (semi) random templates
    templates = []
    for k in range(num_neurons):
        # Space warping using a Gaussian spatial kernel
        center = dist.Uniform(0.0, num_channels).sample()
        width = dist.Uniform(1.0, 1.0 + num_channels / 10.0).sample()
        spatial_factor = torch.exp(
            -0.5 * (torch.arange(num_channels) - center)**2 / width**2)

        # Time warping using a Gaussian temporal kernel
        dt = torch.arange(len_waveform)
        period = len_waveform / (dist.Uniform(1.0, 2.0).sample())
        z = (dt - 0.75 * period) / (.25 * period)
        warp = lambda x: torch.exp(-x)
        window = torch.exp(-0.5 * z**2)
        shape = torch.sin(2 * torch.pi * dt / period)
        temporal_factor = warp(window * shape)

        # Compute the target (outer product of spatial and temporal factors)
        template = torch.outer(spatial_factor, temporal_factor)
        template /= torch.linalg.norm(template)
        templates.append(template)

    return torch.stack(templates)


def generate(num_timesteps,
             num_channels,
             len_waveform,
```

```
          num_neurons,
          mean_amplitude=15,
          shape_amplitude=3.0,
          noise_std=1,
          sample_freq=1000):
    """Create a random set of model parameters and sample data.

    Parameters:
    num_timesteps: integer number of time samples in the data
    num_channels: integer number of channels
    len_waveform: integer duration (number of samples) of each template
    num_neurons: integer number of neurons
    """
    # Make semi-random templates
    templates = generate_templates(num_channels, len_waveform, num_neurons)

    # Make random amplitudes
    amplitudes = torch.zeros((num_neurons, num_timesteps))
    for k in range(num_neurons):
        num_spikes = dist.Poisson(num_timesteps / sample_freq * 10.0).sample()
        sample_shape = (1 + int(num_spikes),)
        times = dist.Categorical(
            torch.ones(num_timesteps) / num_timesteps)\
            .sample(sample_shape)
        amps = dist.Gamma(
            shape_amplitude, shape_amplitude / mean_amplitude)\
            .sample(sample_shape)
        amplitudes[k, times] = amps

        # Only keep spikes separated by at least D
        times, props = find_peaks(amplitudes[k], distance=len_waveform,
                                  height=1e-3)
        amplitudes[k] = 0
        amplitudes[k, times] = torch.tensor(props['peak_heights'],
                                            dtype=torch.float32)

    # Convolve the signal with each row of the multi-channel template
    data = F.conv1d(amplitudes.unsqueeze(0),
                    templates.permute(1, 0, 2).flip(dims=(2,)),
                    padding=len_waveform-1)[0, :, :-(len_waveform-1)]

    data += dist.Normal(0.0, noise_std).sample(data.shape)

    return templates, amplitudes, data
```

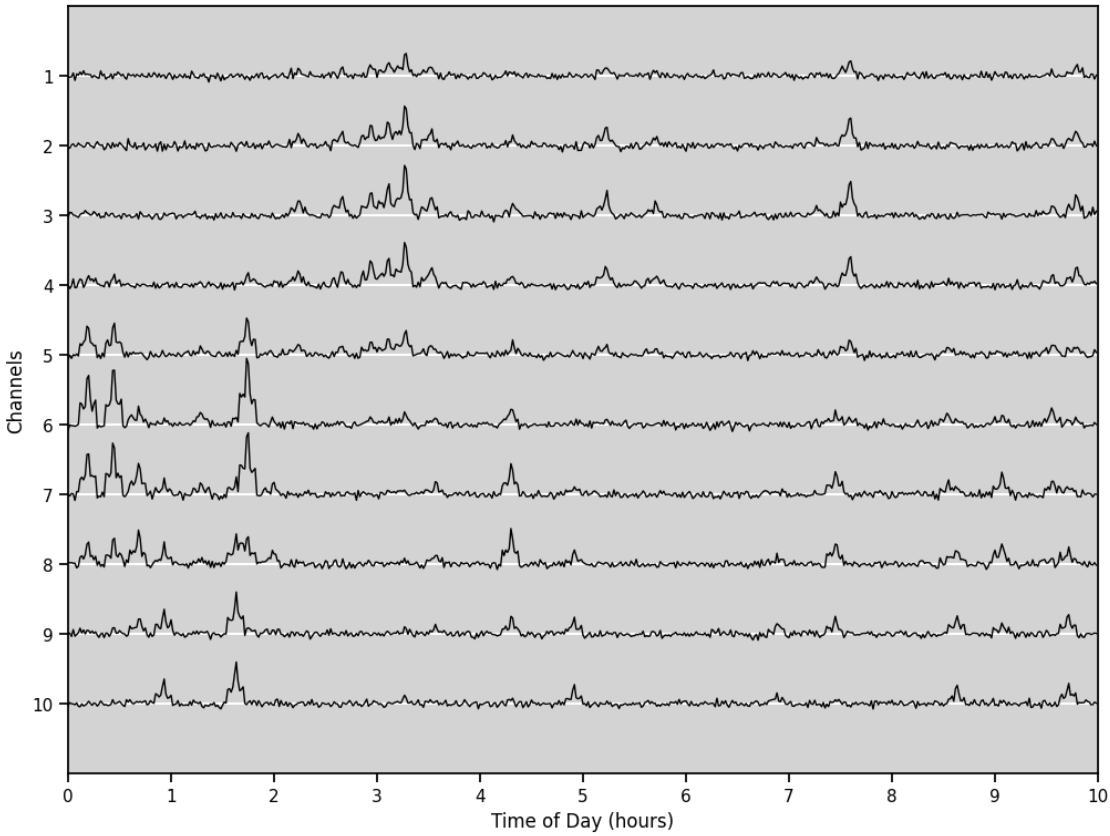
4 Simulate data

By analogy, we examine how this algorithm might be used to recast smartphone behavior, specifically the temporal course of participants' use of 10 applications (N = 10), as a convolution of *spatial wavelets* that indicate qualitatively unique motivations and *temporal amplitudes* that indicate the timing and strength of those motivations. Together, the shape and spiking timing of the **"screen neurons"** reveal when and why the individual is switching among the 10 applications. In this example, we assume there are four (K = 4) ground-truth neurons.

[ ]:
```
# Set hyperparameters
alpha = 3.0
beta = 0.1

T = 600         # number of time samples 10 hours
N = 10          # number of channels
D = 10          # duration of a spike (in samples) ~10 mins
K = 4           # multiple neurons

num_channels = N
num_neurons = K
num_samples = T
spike_width = D

# Generate random templates, amplitudes, and noisy data.
# `templates` are NxCxD and `amplitudes` are NxT
random_seed = 7
torch.manual_seed(random_seed)
print("Simulating data. This could take a few seconds!")
true_templates, true_amplitudes, data = generate(T, N, D, K,
                                                 shape_amplitude=alpha,
                                                 mean_amplitude=alpha/beta)
```

```
Simulating data. This could take a few seconds!
```

[ ]:
```
timestamps = torch.arange(num_samples+1)
plot_data(timestamps, data,
          scale=1.05 * abs(data).max(), plot_slice=slice(0,600))
```



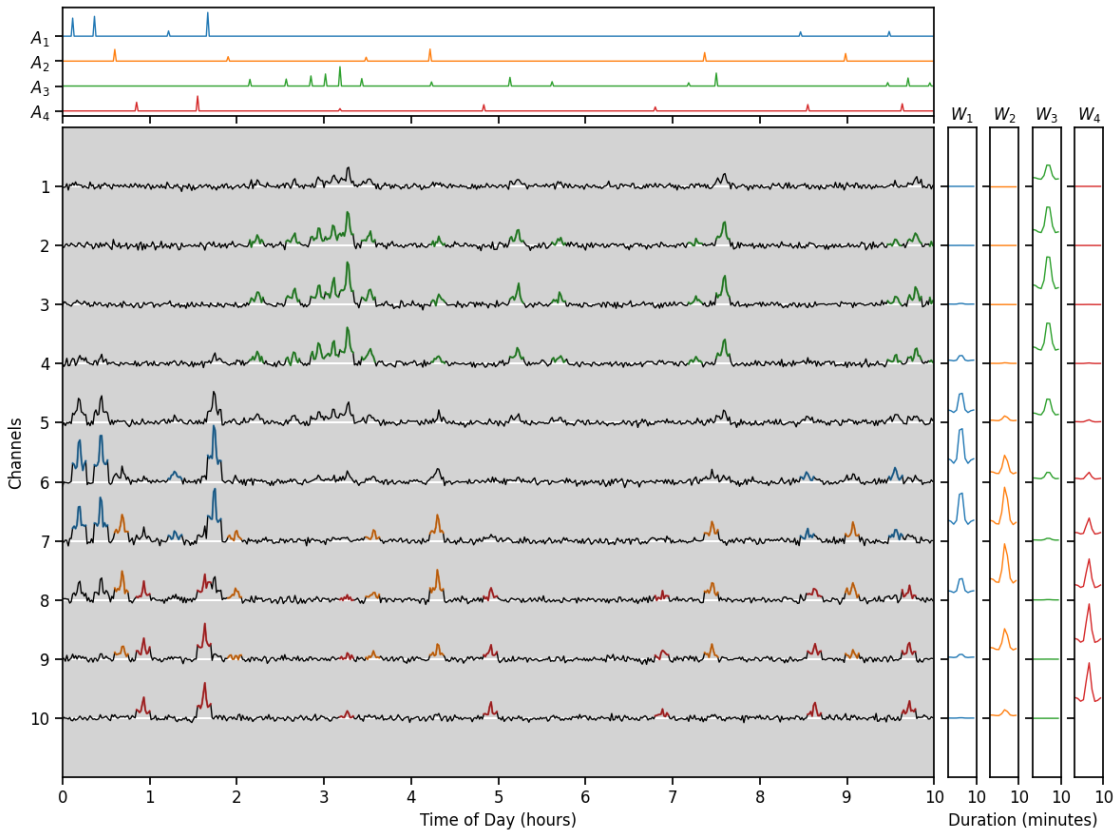
- Interpreted as a neural time-series graph, such as an EEG graph, the time-series indicate how neuronal activity under different areas of the scalp/brain fluctuates over time as an individual engages in different tasks.
- Interpreted from a screenomics perspective, the time-series graph describes when and for how long an individual engaged each of 10 unique dimensions (channels) of media on their smartphone screen.
- In both perspectives, these time series data are decomposed into a smaller set of spatial- and temporal factors that indicate how the underlying motivations and goals that drive the observed behavior change over time.

4.1 Results 1: True motivational states

- The right panel depicts the overlay of 4 ground-truth neurons that were used to generate the data. This panel showcases the spatial waveforms W of these neurons, assuming that the duration of motivational states is under 10 minutes ($D = 10$).
- The top panel showcases their temporal amplitudes A that are associated with contexts of media use. Each color corresponds to a neuron, overlaid when the neuron's spatial wavelets are significantly higher than zero (using top 95th percentile).

[ ]:
```
# Find the spike times and neuron labels
spike_times = []
labels = []
for neuron in trange(num_neurons):
    times = torch.where(true_amplitudes[neuron] > 0)[0]
    spike_times.append(times)
    labels.append(neuron * torch.ones(len(times), dtype=int))

spike_times = torch.cat(spike_times)
labels = torch.cat(labels)

# Find the channels that are significantly modulated in each template
neuron_channels = []
thresh = torch.quantile(abs(true_templates), 0.95)
for template in true_templates:
    neuron_channels.append(torch.any(abs(template) > thresh, axis=1))

plot_model(true_templates, true_amplitudes, data,
           plot_slice=slice(0,600), labels=labels,
           spike_times=spike_times, neuron_channels=neuron_channels)
```

```
  0%|          | 0/4 [00:00<?, ?it/s]
```



5 Fit the model

Previous work has emphasized the connection between task switching and human motivation, studying what/how latent motives are activated by features of media contexts (Wang & Tchernev, 2012; Yeykelis et al., 2014). Two lines of research have emerged to theorize psychological motivations: *uses and gratifications*, and *motivated cognition*.

- Focusing on the internal sources of psychological needs, uses and gratifications (UG) theory describes how individuals select media to fulfill their personal needs or dispositional motives. Thus, researchers attempt to identify the **four** underlying motivations.
- With complementary focus on the physiological activation by external factors, the theory of motivated cognition (MC) suggests how **two** motivational systems, appetitive (approach positive messages) and aversive (avoid negative messages), are activated by media contents.

Thus, we will fit the model using **two different numbers (4 vs. 2)** of screen neurons.

5.1 Results 2: Inferred motivational states (using 4 neurons)

- From the UG perspective, the right panel depicts the overlay of 4 inferred neurons using spike sorting techniques.

[ ]:
```
# Initialize the templates randomly
torch.manual_seed(random_seed)
templates = generate_templates(num_channels, spike_width, num_neurons)

# Copy to GPU
templates = templates.to(device)
amplitudes = torch.zeros(num_neurons, num_samples, device=device,
                         dtype=torch.float32)
data = data.to(device)

# Fit the model
amp_rate = 3.0
noise_std = data.std().item()
lls = map_estimate_fast(templates, amplitudes, data,
                        amp_rate=amp_rate, noise_std=noise_std)
```

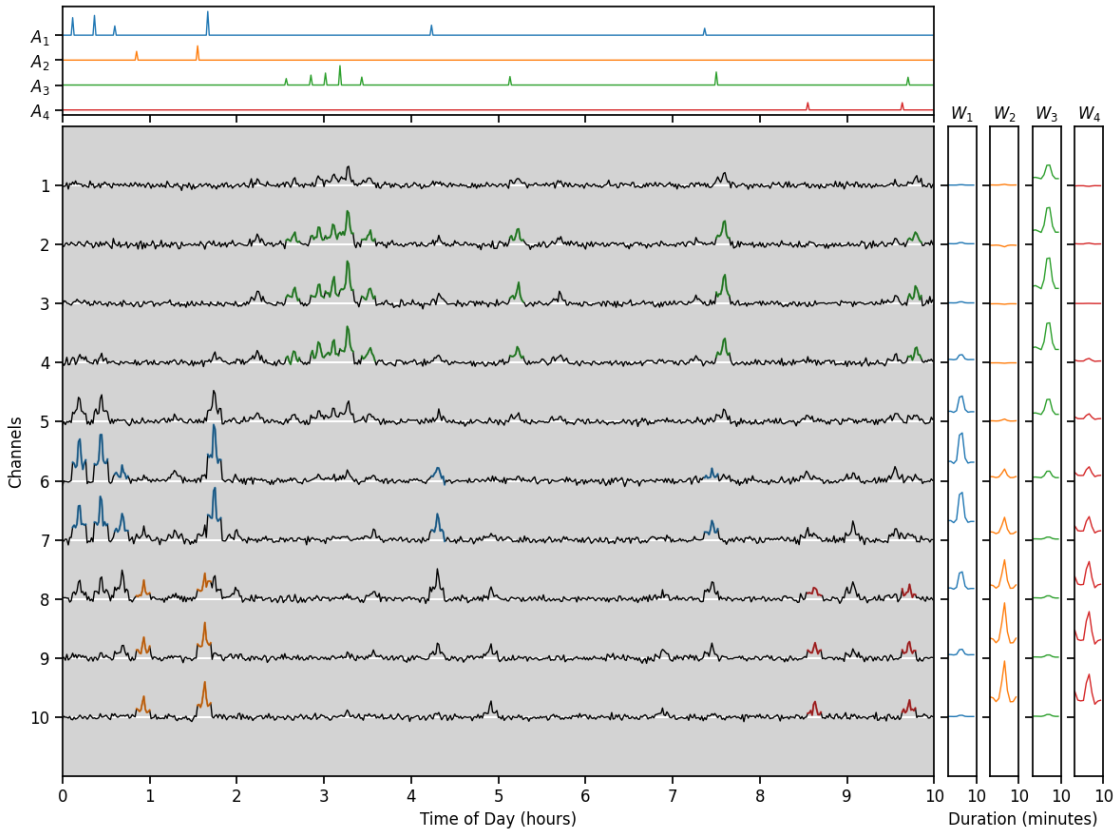
```
  0%|          | 0/20 [00:00<?, ?it/s]
  0%|          | 0/4 [00:00<?, ?it/s]
Convergence detected!
```

[ ]:
```
# Find the spike times and neuron labels
spike_times = []
labels = []
for neuron in trange(num_neurons):
```

```
    times = torch.where(amplitudes[neuron] > 0)[0]
    spike_times.append(times)
    labels.append(neuron * torch.ones(len(times), dtype=int))

spike_times = torch.cat(spike_times)
labels = torch.cat(labels)

# Find the channels that are significantly modulated in each template
neuron_channels = []
thresh = torch.quantile(abs(templates), 0.95)
for template in templates:
    neuron_channels.append(torch.any(abs(template) > thresh, axis=1))

print("Found {} putative spikes (R2 = {:.2f})".format(
    len(spike_times),calculate_r2(data, templates, amplitudes)))
plot_model(templates, amplitudes, data,
           plot_slice=slice(0,600), labels=labels,
           spike_times=spike_times, neuron_channels=neuron_channels)
```

```
  0%|          | 0/4 [00:00<?, ?it/s]
```

```
Found 18 putative spikes (R2 = 0.75)
```



5.2 Results 3: Inferred motivational states (using 2 neurons)

- From the MC perspective, the right panel depicts the overlay of 2 inferred neurons using spike sorting techniques.

[ ]:
```
# Change the number of neurons
num_neurons = 2

# Initialize the templates randomly
torch.manual_seed(random_seed)
templates = generate_templates(num_channels, spike_width, num_neurons)

# Copy to GPU
templates = templates.to(device)
amplitudes = torch.zeros(num_neurons, num_samples, device=device,
                         dtype=torch.float32)
data = data.to(device)

# Fit the model
amp_rate = 3.0
noise_std = data.std().item()
lls = map_estimate_fast(templates, amplitudes, data,
                        amp_rate=amp_rate, noise_std=noise_std)
```

```
  0%|          | 0/20 [00:00<?, ?it/s]
  0%|          | 0/2 [00:00<?, ?it/s]
Convergence detected!
```

[ ]:
```
# Find the spike times and neuron labels
spike_times = []
labels = []
for neuron in trange(num_neurons):
    times = torch.where(amplitudes[neuron] > 0)[0]
    spike_times.append(times)
    labels.append(neuron * torch.ones(len(times), dtype=int))

spike_times = torch.cat(spike_times)
labels = torch.cat(labels)

# Find the channels that are significantly modulated in each template
neuron_channels = []
thresh = torch.quantile(abs(templates), 0.95)
for template in templates:
    neuron_channels.append(torch.any(abs(template) > thresh, axis=1))

print("Found {} putative spikes (R2 = {:.2f})".format(
    len(spike_times),calculate_r2(data, templates, amplitudes)))
```

```
plot_model(templates, amplitudes, data,
           plot_slice=slice(0,600), labels=labels,
           spike_times=spike_times, neuron_channels=neuron_channels)
```

```
  0%|          | 0/2 [00:00<?, ?it/s]
Found 10 putative spikes (R2 = 0.44)
```