Pynapple & NeMoS Cheat Sheet

`[pynapple]` Neural data structures `[NeMoS]` Neural modeling with GLMs

Pynapple Data Types

Core Types

- **Ts**: Timestamps (event times, e.g., spikes)
- **Tsd**: Time series data (1D with time)
- **TsdFrame**: Tabular time series (2D)
- **TsdTensor**: Multi-dimensional time series
- **TsGroup**: Collection of Ts objects
- **IntervalSet**: Time intervals (epochs, trials)

Initialization

```
import numpy as np
import pynapple as nap

t = np.linspace(0, 5, 100)

# 1D time series
tsd1 = nap.Tsd(t=t, d=np.randn(100))

# 2D time series (tabular)
tsd2 = nap.TsdFrame(t=t, d=np.randn(100, 3),
    columns=["a", "b", "c"])

# Epochs/intervals
epochs = nap.IntervalSet(start=[1, 10],
    end=[2, 15])

# Spike trains
ts0 = np.sort(np.random.uniform(0, 5, 200))
ts1 = np.sort(np.random.uniform(0, 5, 200))
spikes = nap.TsGroup({0: ts0, 1: ts1})

# Add metadata to TsGroup
spikes["area"] = ["a1", "mso"]
spikes.set_info(channel=[1, 2])
```

Data Selection & Indexing

Indexing (NumPy-style)

```
# Scalar index -> NumPy array
ts0[1]              # Time of 2nd event
tsd2[0]             # First row

# List index -> Pynapple object
ts0[[0,1,2]]        # First 3 events
ts0[:3]             # Same as above

# TsGroup: scalar selects Ts
spikes[0]           # Neuron 0's spikes

# TsGroup: list selects TsGroup
spikes[[0,1]]       # Neurons 0 and 1
```

Time Support

```
# Overall time support
spikes.time_support   # Start to end

# Find support with min gap
bursts = ts0.find_support(10e-3)
```

Restricting Data

```
# Limit to specific intervals
spikes_trial = spikes.restrict(epochs)

tsd_epoch = tsd1.restrict(epochs)
```

Filtering TsGroups

```
# With booleans
slow = spikes[spikes.rate < 10]

# By category
cort = spikes.getby_category("area")["a1"]
```

Data Manipulation

```
# NumPy ufuncs work natively
np.max(tsd1)
np.mean(tsd1)

# Binning spike counts
histograms = spikes.count(0.020)
# -> TsdFrame with 20ms bins

# Downsample by binning (Tsd*)
low_res = tsd1.bin_average(2.0)

# Smooth with Gaussian kernel (Tsd*)
smooth = tsd1.smooth(std=1.0)

# Threshold values (Tsd*)
positive = tsd1.threshold(0.0)

# Interpolate (Tsd*)
velocity = tsd1.interpolate(tsd2)
```

```
# Extract values at specific times
synced = ts0.value_from(tsd1)

# Combine IntervalSet
combined = epochs.union(other_epochs)

# Intersection (IntervalSet)
overlap = epochs.intersect(ts0.time_support)

# Set difference (IntervalSet)
outside = epochs.set_diff(overlap)
```

Analysis Functions

Tuning Curves

```
# 1D tuning curves
tuning = nap.compute_tuning_curves(
    data, features, bins=N
)
```

Decoding

```
# Bayesian decoding
prediction, probs = nap.decode_bayes(
    tuning_curves, data
)

# Template decoding
prediction, probs = nap.decode_template(
    tuning_curves, data
)
```

Perievent Time Histograms

```
# PETH aligned to events
peth = nap.compute_perievent(
    spikes,         # Ts/TsGroup
    stim_times,     # Reference times
    (-1, 3)         # -1s to +3s window
)
```

Wavelet Transform

```
# Compute wavelets
freqs = np.geomspace(10, 100, 20)
wlets = nap.compute_wavelet_transform(
    tsd, freqs, fs=12500
)
```

NeMoS: Neural Modeling

Basic GLM Workflow

```
# Initialize GLM
model = nmo.glm.GLM(solver_name="LBFGS",
    regularizer="Ridge")

# Fit
model.fit(features, count)

# Predict
pred = model.predict(features) # (spk/ bin)
pred *= count.rate # (Hz)

# Score
score = model.score(features, count)
```

Population GLM

```
# Multiple neurons (same arguments as GLM)
pop_model = nmo.glm.PopulationGLM()

# counts shape (n_samples, n_neurons)
pop_model.fit(features, counts)
```

Basis Functions

Convolutional Bases

```
# History basis (autoregressive)
win_size = 80 # bins

hist_basis = nmo.basis.HistoryConv(win_size)

hist_features = hist_basis.compute_features(
    counts
) # -> (n_time, n_neurons * win_size)

# Ten raised cosine log basis with label
cos_basis = nmo.basis.RaisedCosineLogConv(
    n_basis_funcs=10,
    window_size=win_size,
    label="cos"
)

cos_features = cos_basis.compute_features(
    counts
) # -> (n_time, n_neurons * n_basis)

# Fit with features
model = nmo.glm.PopulationGLM()
model.fit(cos_features, counts)
```

Basis Composition

```
position = nmo.basis.BSplineEval(
    5, label="pos"
)
speed = nmo.basis.MSplineEval(
    6, label="speed"
)

# Basis addition (concatenation)
add = position + speed
X = add.compute_features(pos_var, speed_var)
# (n_samples, n_basis1 + n_basis2)

# Basis multiplication (interaction)
mul = position * speed
X = mul.compute_features(pos_var, speed_var)
# (n_samples, n_basis1 * n_basis2)

# Get a basis component
position = add["pos"]
```

Interpreting Weights

```
# Split weights by basis
weights = add.split_by_feature(
    model_basis.coef_,
    axis=0
)
```

scikit-learn Integration

Pipelines

```
from sklearn.pipeline import Pipeline

# Create transformer from basis
transform = basis.to_transformer()
transform.set_input_shape(counts)

# Build pipeline
pipe = Pipeline([
    ("basis", transform),
    ("glm", nmo.glm.GLM()),
])

# Fit and predict
pipe.fit(counts, counts)
pipe.predict(counts)

# Get steps
basis = pipe["basis"]
glm = pipe["glm"]
```

Cross-Validation

```
from sklearn.model_selection import
    GridSearchCV

# Define hyperparameter grid
param_grid = {
    "basis__n_basis_funcs": [5, 10, 15, 20],
    "glm__regularizer": [
        nmo.regularizer.UnRegularized(),
        nmo.regularizer.Ridge()
    ],
}

# Grid search with CV
cv = GridSearchCV(pipe, param_grid, cv=3)
cv.fit(counts, counts)

# Get best fitting model
best_model = cv.best_estimator_

# Get cv results
cv.cv_results_
```

Advanced Grid Search

```
# Conditional hyperparameters
param_grid = [
    # Try unregularized
    {
        "glm__regularizer": [
        nmo.regularizer.UnRegularized()]
    },
    # Try Ridge with strengths
    {
        "glm__regularizer":
        [nmo.regularizer.Ridge()],
        "glm__regularizer_strength":
        [1e-6, 1e-3, 1e-1]
    },
]
```

Key Concepts:

- Pynapple: Time-aware data structures
- NeMoS: sklearn-style GLM interface
- Basis functions reduce parameters
- Pipelines combine preprocessing + modeling
- Cross-validation for hyperparameter tuning