Neural Data Analysis

Lecturer: Jan Lause, Prof. Dr. Philipp Berens

Tutors: Jonas Beck, Rita González Márquez, Fabio Seel

Summer term 2024

Name: Carlotta Trottenberg, Jungmin Lee, Ralf Krüger

Coding Lab 5

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import scipy.optimize as opt
import scipy.io as io
from scipy.special import factorial

#%load_ext jupyter_black

#%load_ext watermark
#%watermark --time --date --timezone --updated --python --iversions --watermark

#%matplotlib inline
#plt.style.use("../matplotlib_style.txt")
```

Task 1: Fit RF on simulated data

We will start with toy data generated from an LNP model neuron to make sure everything works right. The model LNP neuron consists of one Gaussian linear filter, an exponential nonlinearity and a Poisson spike count generator. We look at it in discrete time with time bins of width δt. The model is:

$$c_t \sim Poisson(r_t)$$
$$r_t = \exp(w^T s_t) \cdot \Delta t \cdot R$$

Here, c_t is the spike count in time window t of length Δt, s_t is the stimulus and w is the receptive field of the neuron. The receptive field variable `w` is 15 × 15 pixels and normalized to $||w|| = 1$. A stimulus frame is a 15 × 15 pixel image, for which we use uncorrelated checkerboard noise. R can be used to bring the firing rate into the right regime (e.g. by setting $R = 50$).

For computational ease, we reformat the stimulus and the receptive field in a 225 by 1 array. The function `sample_lnp` can be used to generate data from this model. It returns a spike count vector `c` with samples from the model (dimensions: 1 by nT = $T/\Delta t$), a stimulus matrix `s` (dimensions: 225 × nT) and the mean firing rate `r` (dimensions: nT × 1).

Here we assume that the receptive field influences the spike count instantaneously just as in the above equations. Implement a Maximum Likelihood approach to fit the receptive field.

To this end simplify and implement the log-likelihood function $L(w)$ and its gradient $\frac{L(w)}{dw}$ with respect to w (`negloglike_lnp`). The log-likelihood of the model is

$$L(w) = \log \prod_t \frac{r_t^{c_t}}{c_t!} \exp(-r_t).$$

Plot the true receptive field, a stimulus frame, the spike counts and the estimated receptive field.

Grading: 2 pts (calculations) + 3 pts (implementation)

Calculations

You can add your calculations in LaTeX here.

$$L(\omega) = \log \prod_t \frac{r_t^{c_t}}{c_t!} \exp(-r_t)$$

Log of a product is the sum of the logs:

$$L(\omega) = \sum_t (\log(\frac{r_t^{c_t}}{c_t!} \exp(-r_t)))$$

Log of a quotient is the difference of the logs:

$$L(\omega) = \sum_t (\log(r_t^{c_t}) - \log(c_t!) + \log(\exp(-r_t)))$$

$$= \sum_t (c_t \cdot \log(r_t) - \log(c_t!) - r_t)$$

The negative log-likelihood is thus:

$$= \sum_t (-c_t \cdot \log(r_t) + \log(c_t!) + r_t)$$

Differentiate $L(\omega)$: $\log(c_t!)$ is a constant with respect to ω and can thus be dropped:

$$\frac{dL(\omega)}{d\omega} = \sum_t (\frac{d}{d\omega}(-c_t \cdot \log(r_t)) + \frac{d}{d\omega}(r_t))$$

Plug in $r_t = \exp(w^T s_t) \cdot \Delta t \cdot R$:

$$\frac{dL(\omega)}{d\omega} = \sum_t (\frac{d}{d\omega}(-c_t \cdot \log(\exp(w^T s_t) \cdot \Delta t \cdot R)) + \frac{d}{d\omega}(\exp(w^T s_t) \cdot \Delta t \cdot R))$$

$$= \sum_t (\frac{d}{d\omega}(-c_t \cdot w^T s_t) + \frac{d}{d\omega}(\exp(w^T s_t) \cdot \Delta t \cdot R))$$

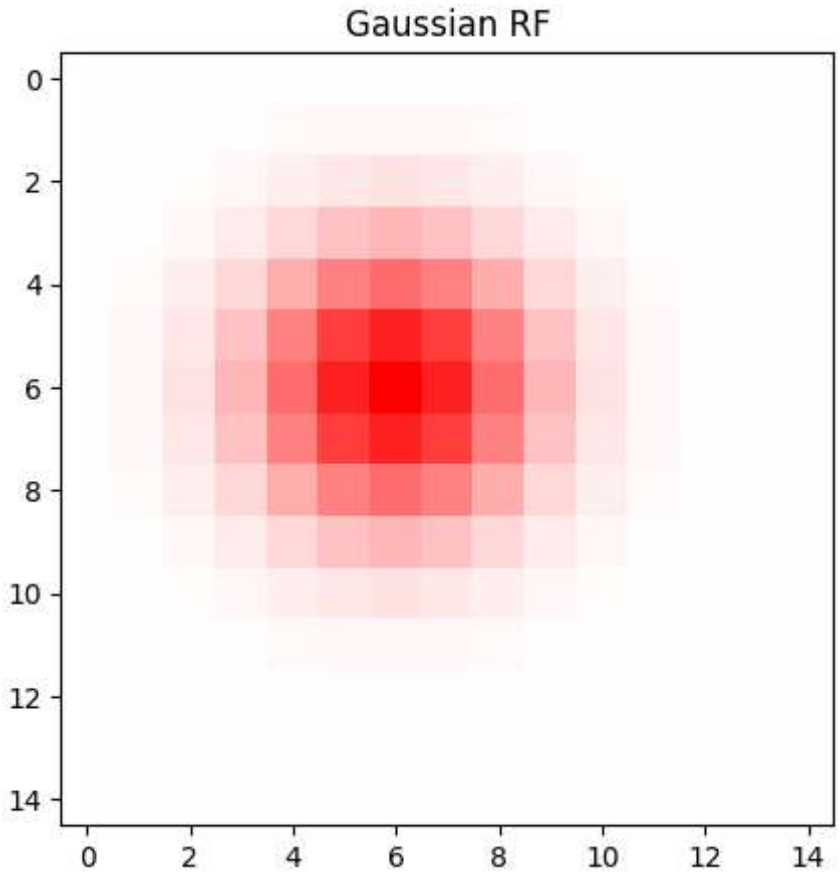
$$= \sum_t (-c_t s_t + s_t \cdot \exp(w^T s_t) \cdot \Delta t \cdot R)$$

$$= \sum_t (s_t(-c_t + r_t))$$

Generate data

In [ ]:
```python
def gen_gauss_rf(D: int, width: float, center: tuple = (0, 0)) -> np.ndarray:
    """
    Generate a Gaussian receptive field.

    Args:
        D (int): Size of the receptive field (DxD).
        width (float): Width parameter of the Gaussian.
        center (tuple, optional): Center coordinates of the receptive field. Def

    Returns:
        np.ndarray: Gaussian receptive field.
    """

    sz = (D - 1) / 2
    x, y = np.meshgrid(np.arange(-sz, sz + 1), np.arange(-sz, sz + 1))
    x = x + center[0]
    y = y + center[1]
    w = np.exp(-(x**2 / width + y**2 / width))
    w = w / np.sum(w.flatten())

    return w


w = gen_gauss_rf(15, 7, (1, 1))

vlim = np.max(np.abs(w))
fig, ax = plt.subplots(1, 1, figsize=(5, 5))
ax.imshow(w, cmap="bwr", vmin=-vlim, vmax=vlim)
ax.set_title("Gaussian RF")
```

Out[ ]:
```
Text(0.5, 1.0, 'Gaussian RF')
```

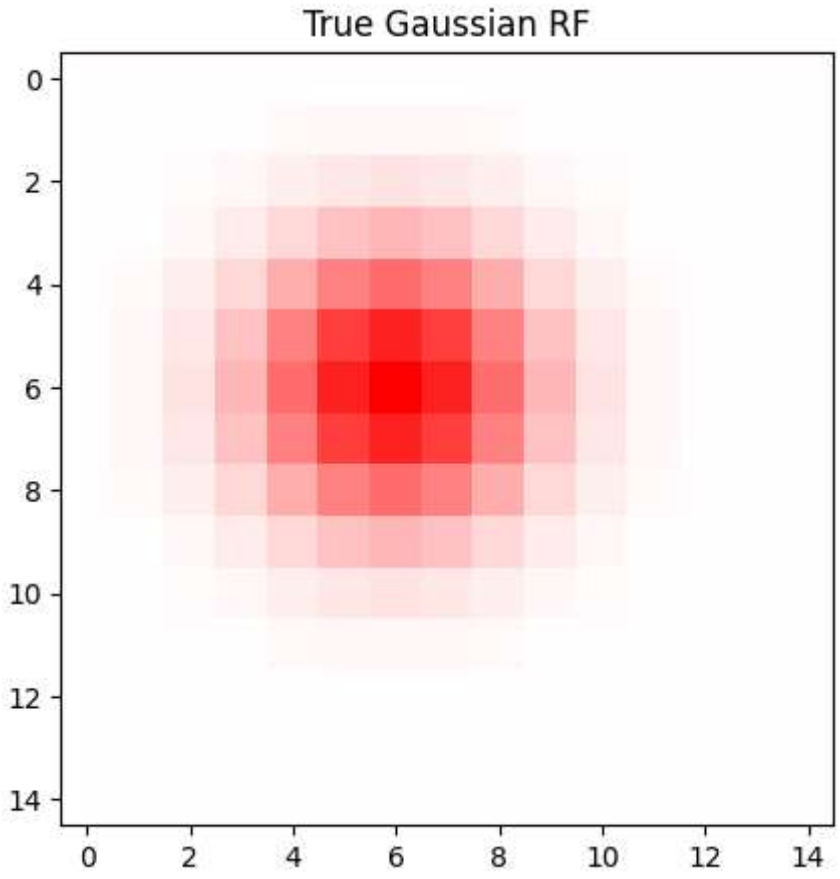


In [ ]:
```python
def sample_lnp(
    w: np.array, nT: int, dt: float, R: float, v: float, random_seed: int = 10
):
    """Generate samples from an instantaneous LNP model neuron with
    receptive field kernel w.

    Parameters
    ----------

    w: np.array, (Dx * Dy, )
        (flattened) receptive field kernel.

    nT: int
        number of time steps

    dt: float
        duration of a frame in s

    R: float
        rate parameter

    v: float
        variance of the flattened stimulus array

    random_seed: int
        seed for random number generator

    Returns
    -------

    c: np.array, (nT, )
        sampled spike counts in time bins
```

```
    r: np.array, (nT, )
        mean rate in time bins

    s: np.array, (Dx * Dy, nT)
        stimulus frames used

    Note
    ----

    See equations in task description above for a precise definition
    of the individual parameters.

    """

    rng = np.random.default_rng(random_seed)

    # insert your code here

    # ------------------------------------------------
    # Generate samples from an instantaneous LNP model
    # neuron with receptive field kernel w. (0.5 pts)
    # ------------------------------------------------

    # set seed
    rng = np.random.default_rng(random_seed)
    w_len = w.shape[0]

    # generate samples: uncorrelated checkerboard noise as stims
    s = rng.normal(0, np.sqrt(v), (w_len, nT))

    # Compute the linear filter response
    wTs = w.T @ s

    # Compute the mean firing rate
    r = np.exp(wTs) * dt * R

    # Generate spike counts using Poisson distribution
    c = rng.poisson(r)

    return c, r, s
```

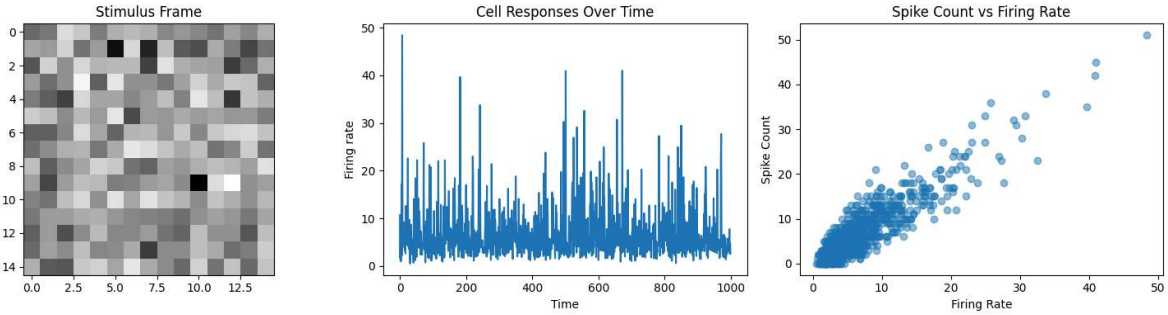
In [ ]:
```
D = 15  # number of pixels
nT = 1000  # number of time bins
dt = 0.1  # bins of 100 ms
R = 50  # firing rate in Hz
v = 20  # stimulus variance

w = gen_gauss_rf(D, 7, (1, 1))
w = w.flatten()

c, r, s = sample_lnp(w, nT, dt, R, v)
```

In [ ]:
```
# Plot the true receptive field
vlim = np.max(np.abs(w))
fig, ax = plt.subplots(1, 1, figsize=(5, 5))
ax.imshow(w.reshape((D, D)), cmap="bwr", vmin=-vlim, vmax=vlim)
ax.set_title("True Gaussian RF")
plt.show()
```



Plot the stimulus for one frame, the cell's response over time and the spike count vs firing rate.

In [ ]:
```
mosaic = mosaic = [["stim", "responses", "count/rate"]]

fig, ax = plt.subplot_mosaic(mosaic=mosaic, figsize=(15, 4))
# ------------------------------------------------------------------------
# Plot the stimulus for one frame, the cell's responses over time and spike coun
# ------------------------------------------------------------------------

# plot the stimulus for one frame == our uncorrelated checkerboard noise
ax["stim"].imshow(s[:, 0].reshape((D, D)), cmap="gray")
ax["stim"].set_title("Stimulus Frame")

# cell's responses over time
#ax["responses"].plot(c, label='Spike Counts')
ax["responses"].plot(r, label='Firing Rate')
ax["responses"].set_title("Cell Responses Over Time")
#ax["responses"].legend()
ax["responses"].set_xlabel('Time')
ax["responses"].set_ylabel("Firing rate")


# spike count vs FR
ax["count/rate"].scatter(r, c, alpha=0.5)
ax["count/rate"].set_title("Spike Count vs Firing Rate")
ax["count/rate"].set_xlabel("Firing Rate")
ax["count/rate"].set_ylabel("Spike Count")

plt.tight_layout()
plt.show()
```



Implementation

Implement the negative log-likelihood of the LNP and its gradient with respect to the receptive field using the simplified equations you calculated earlier *(0.5 pts)*

In [ ]:
```python
def negloglike_lnp(
    w: np.array, c: np.array, s: np.array, dt: float = 0.1, R: float = 50
) -> float:
    """Implements the negative (!) log-likelihood of the LNP model

    Parameters
    ----------

    w: np.array, (Dx * Dy, )
      current receptive field

    c: np.array, (nT, )
      spike counts

    s: np.array, (Dx * Dy, nT)
      stimulus matrix


    Returns
    -------

    f: float
      function value of the negative log likelihood at w

    """

    # ------------------------------------------------
    # Implement the negative log-likelihood of the LNP
    # ------------------------------------------------

    # define most important bits:
    w = w.reshape(-1, 1) # column vector
    wTs = w.T @ s
    r = np.exp(wTs) * dt * R

    # Compute negative log-likelihood
    f = np.sum(-c * np.log(r) + np.log(factorial(c)) + r)

    # insert your code here
    return f


def deriv_negloglike_lnp(
```

```
    w: np.array, c: np.array, s: np.array, dt: float = 0.1, R: float = 50
) -> np.array:
    """Implements the gradient of the negative log-likelihood of the LNP model

    Parameters
    ----------

    see negloglike_lnp

    Returns
    -------

    df: np.array, (Dx * Dy, )
      gradient of the negative log likelihood with respect to w

    """

    # ---------------------------------------------------------------
    # Implement the gradient with respect to the receptive field `w`
    # ---------------------------------------------------------------

    w = w.reshape(-1, 1) # column vector
    wTs = w.T @ s
    r = np.exp(wTs) * dt * R


    df = ((-c + r) @ s.T)

    return df
```

The helper function `check_grad` in `scipy.optimize` can help you to make sure your equations and implementations are correct. It might be helpful to validate the gradient before you run your optimizer.

In [ ]:
```
# Check gradient
# insert your code here
from scipy.optimize import check_grad
w = w.flatten()
error = check_grad(negloglike_lnp, deriv_negloglike_lnp, w, c, s, dt, R)
print("Gradient check error:", error)
```

```
Gradient check error: 0.015141313568195206
```

Fit receptive field maximizing the log likelihood.

The scipy.optimize package also has suitable functions for optimization. If you generate a large number of samples, the fitted receptive field will look more similar to the true receptive field. With more samples, the optimization takes longer, however.

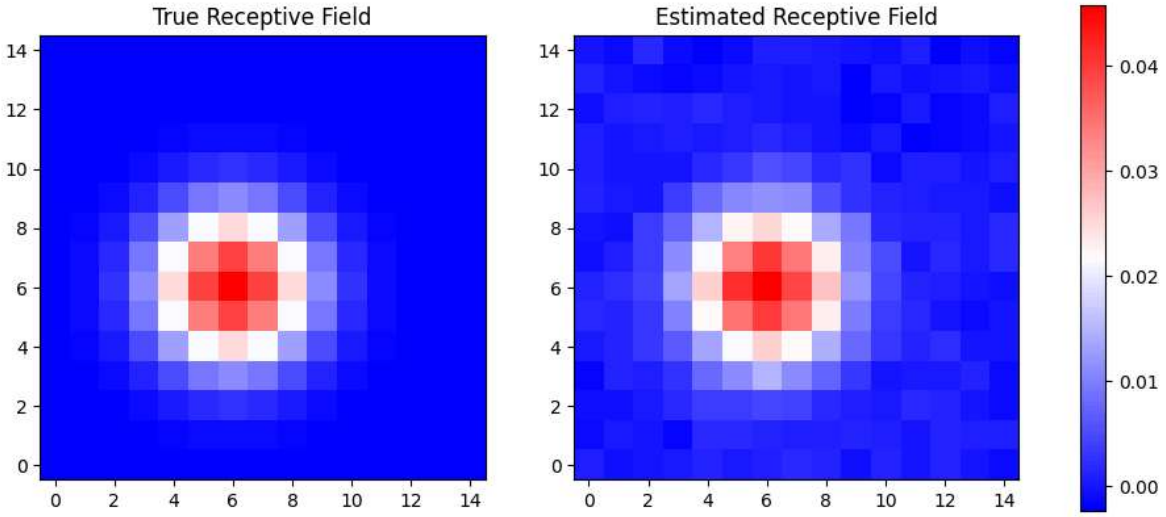
In [ ]:
```
# ----------------------------------------
# Estimate the receptive field by maximizing
# the log-likelihood (or more commonly,
# minimizing the negative log-likelihood).
#
# Tips: use scipy.optimize.minimize(). (1 pt)
# ----------------------------------------

rng = np.random.default_rng(1234)
```

```
from scipy.optimize import minimize

D = 15  # number of pixels
nT = 10000  # number of time bins
dt = 0.1  # bins of 100 ms
R = 50  # firing rate in Hz
v = 20  # stimulus variance

w = gen_gauss_rf(D, 7, (1, 1))
w = w.flatten()

c, r, s = sample_lnp(w, nT, dt, R, v)

# Initial guess for the rf
w_init = rng.normal(0, 1, D*D)

# Perform the optimization
result = minimize(
    fun=negloglike_lnp,
    x0=w_init,
    args=(c, s, dt, R),
    jac=deriv_negloglike_lnp,
    method='L-BFGS-B'
)

# Extract the estimated receptive field
w_est = result.x

# did it work?
print("Optimization success?? ", result.success)
```

```
Optimization success??  True
```

In [ ]:

```
# insert your code here

# ------------------------------------
# Plot the ground truth and estimated
# `w` side by side. (0.5 pts)
# ------------------------------------

mosaic = [["True", "Estimated"]]
fig, ax = plt.subplot_mosaic(mosaic=mosaic, figsize=(12, 5))

# make sure to add a colorbar. 'bwr' is a reasonable choice for the cmap.

# Reshape the true and estimated receptive fields for visualization
w_true_reshaped = w.reshape(D, D)
w_est_reshaped = w_est.reshape(D, D)

# Visualize the true and estimated receptive fields
ax["True"].set_title("True Receptive Field")
true_rf = ax["True"].imshow(w_true_reshaped, cmap='bwr', origin='lower')

ax["Estimated"].set_title("Estimated Receptive Field")
est_rf = ax["Estimated"].imshow(w_est_reshaped, cmap='bwr', origin='lower')

cbar = fig.colorbar(est_rf, ax=[ax["True"], ax["Estimated"]], orientation='verti

plt.show()
```



Task 2: Apply to real neuron

Download the dataset for this task from Ilias (`nds_cl_5_data.mat`). It contains a stimulus matrix (`s`) in the same format you used before and the spike times. In addition, there is an array called `trigger` which contains the times at which the stimulus frames were swapped.

- Generate an array of spike counts at the same temporal resolution as the stimulus frames
- Fit the receptive field with time lags of 0 to 4 frames. Fit them one lag at a time (the ML fit is very sensitive to the number of parameters estimated and will not produce good results if you fit the full space-time receptive field for more than two time lags at once).
- Plot the resulting filters

Grading: 2 pts

In [ ]:
```
var = io.loadmat("../data/nds_cl_5_data.mat")

# t contains the spike times of the neuron
t = var["DN_spiketimes"].flatten()

# trigger contains the times at which the stimulus flipped
trigger = var["DN_triggertimes"].flatten()

# contains the stimulus movie with black and white pixels
s = var["DN_stim"]
s = s.reshape((300, 1500))  # the shape of each frame is (20, 15)
s = s[:, 1 : len(trigger)]
```

Create vector of spike counts

In [ ]:
```
# insert your code here

# ----------------------------------------
# Bin the spike counts at the same temporal
# resolution as the stimulus (0.5 pts)
# ----------------------------------------
```

```
# shape of s = 300,1488
# shape of trigger = 1489
# shape of t = 5500

# as the number of trigger times is one larger than the number of stimuli, I ass
# thus trigger gives us the needed bin edges for binning the spikes

binned_spike_counts, _ = np.histogram(t,trigger)
binned_spike_counts
```

Out[ ]:
```
array([6, 6, 9, ..., 0, 0, 0], dtype=int64)
```

In [ ]:
```
```

Fit receptive field for each frame separately

In [ ]:
```
# insert your code here

# ----------------------------------------
# Fit the receptive field with time lags of
# 0 to 4 frames separately (1 pt)
#
# The final receptive field (`w_hat`) should
# be in the shape of (Dx * Dy, 5)
# ----------------------------------------

# specify the time lags
delta = [0, 1, 2, 3, 4]

# fit for each delay

Dx = 20
Dy = 15
dt = 0.1
R = 50

# init w_hat
w_hat = np.zeros((Dx * Dy, len(delta)))

# Fit the receptive field for each time lag
for i, lag in enumerate(delta):

    if lag > 0:
        # remove last few stimulus frames according to lag (e.g. last 4 frames)
        # take the counts from "lag" onwards:
        # now, the "matching" stimulus will lag behind, and spike counts will be
        c_shifted = binned_spike_counts[lag:]
        s_shifted = s[:, :-lag]
    else:
        c_shifted = binned_spike_counts
        s_shifted = s

    # Initial guess for the receptive field
    w_init = rng.normal(0, 1, Dx *Dy)

    # Perform the optimization
    result = minimize(
        fun=negloglike_lnp,
```

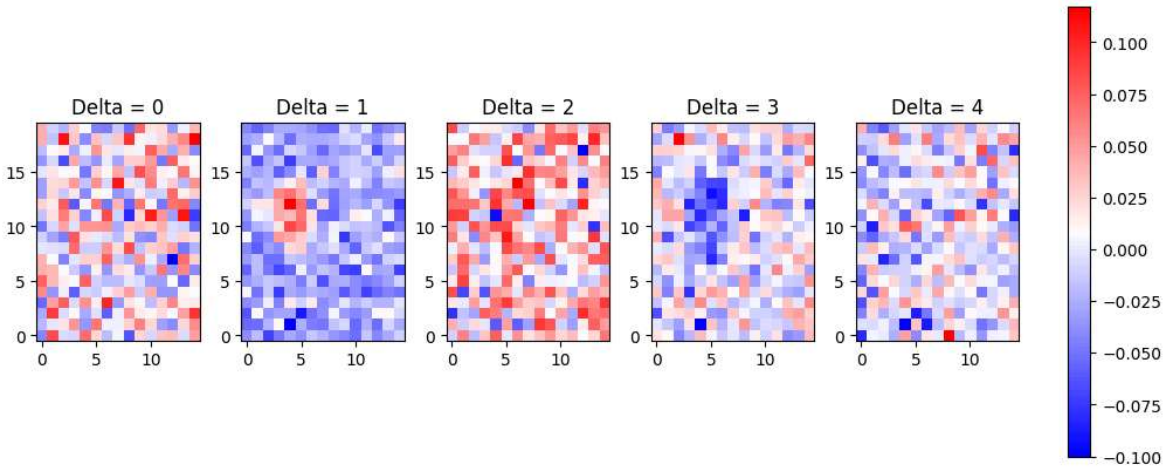
```
        x0=w_init,
        args=(c_shifted, s_shifted, dt, R),
        jac=deriv_negloglike_lnp,
        method='L-BFGS-B'
    )

    # Store the estimated receptive field
    w_hat[:, i] = result.x

# did it work?
print("Optimization success:", result.success)
```

```
Optimization success: True
```

Plot the frames one by one

In [ ]:

```
# insert your code here

# ----------------------------------------
# Plot all 5 frames of the fitted RFs (0.5 pt)
# ----------------------------------------

fig, ax = plt.subplot_mosaic(mosaic=[delta], figsize=(10, 4), constrained_layout

for lag in delta:

    ax[lag].set_title(f"Delta = {lag}")
    w_reshaped = w_hat[:,lag].reshape(Dx,Dy)
    rf = ax[lag].imshow(w_reshaped, cmap='bwr', origin='lower')

cbar = fig.colorbar(rf, ax=[ax[0], ax[1], ax[2], ax[3], ax[4]],  orientation='ve

plt.show()
```



Task 3: Separate space/time components

The receptive field of the neuron can be decomposed into a spatial and a temporal component. Because of the way we computed them, both are independent and the resulting spatio-temporal component is thus called separable. As discussed in the lecture, you can use singular-value decomposition to separate these two:

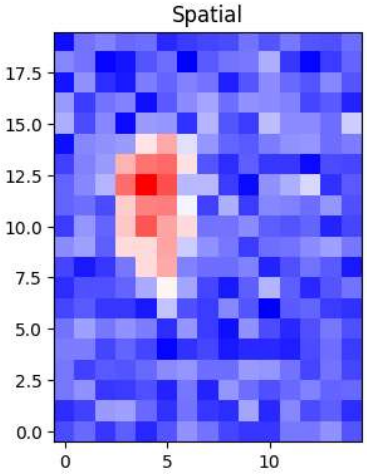
$$W = u_1 s_1 v_1^T$$

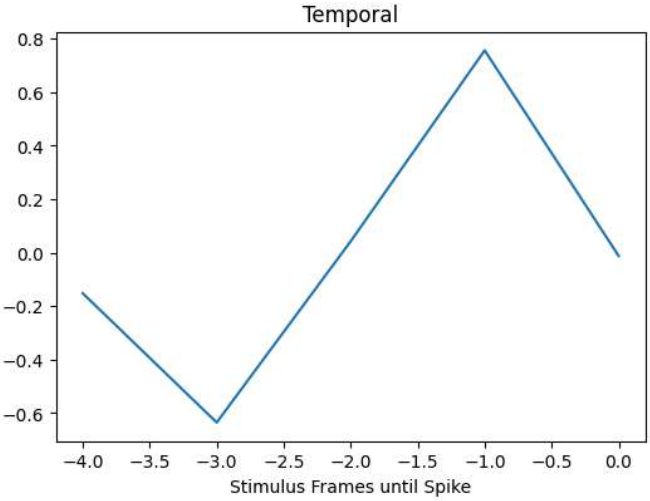
Here u_1 and v_1 are the singular vectors belonging to the 1st singular value s_1 and provide a long rank approximation of W, the array with all receptive fields. It is important that the mean is subtracted before computing the SVD.

Plot the first temporal component and the first spatial component. You can use a Python implementation of SVD. The results can look a bit puzzling, because the sign of the components is arbitrary.

Grading: 1 pts

In [ ]:
```
# insert your code here

# ------------------------------------------
# Apply SVD to the fitted receptive field,
# you can use either numpy or sklearn (0.5 pt)
# ------------------------------------------

# shape of w_hat: (300,5).
# subtract mean along axis=1 (time) since for every timelag the RF is estimated
w_mean_corrected = w_hat - np.mean(w_hat, axis=1).reshape((300, 1))

from sklearn.utils.extmath import randomized_svd

u1, s1, v1 = randomized_svd(w_mean_corrected, n_components=1)

# reshape u1 into a matrix
u1_reshaped = u1.reshape(Dx,Dy)


# ------------------------------------------------
# Plot the spatial and temporal components (0.5 pt)
# ------------------------------------------------

fig, ax = plt.subplot_mosaic(
    mosaic=[["Spatial", "Temporal"]], figsize=(10, 4), constrained_layout=True
)
# add plot
ax["Spatial"].set_title(f"Spatial")
rf = ax["Spatial"].imshow(u1_reshaped, cmap='bwr', origin='lower')

ax["Temporal"].set_title(f"Temporal")
ax["Temporal"].plot((-4,-3,-2,-1,0),v1[0,::-1])
ax["Temporal"].set_xlabel('Stimulus Frames until Spike')
plt.show()
```





Task 4: Regularized receptive field

As you can see, maximum likelihood estimation of linear receptive fields can be quite noisy, if little data is available.

To improve on this, one can regularize the receptive field vector and a term to the cost function

$$C(w) = L(w) + \alpha||w||_p^2$$

Here, the p indicates which norm of w is used: for $p = 2$, this is shrinks all coefficient equally to zero; for $p = 1$, it favors sparse solutions, a penality also known as lasso. Because the 1-norm is not smooth at zero, it is not as straightforward to implement "by hand".

Use a toolbox with an implementation of the lasso-penalization and fit the receptive field. Possibly, you will have to try different values of the regularization parameter α. Plot your estimates from above and the lasso-estimates. How do they differ? What happens when you increase or decrease $alpha$?

If you want to keep the Poisson noise model, you can use the implementation in `pyglmnet`. Otherwise, you can also resort to the linear model from `sklearn` which assumes Gaussian noise (which in my hands was much faster).

Grading: 2 pts

In [ ]:
```
def negloglike_lnp_L1(
    w: np.array, c: np.array, s: np.array, dt: float = 0.1, R: float = 50
) -> float:
    """Implements the negative (!) log-likelihood of the LNP model

    Parameters
    ----------

    w: np.array, (Dx * Dy, )
      current receptive field

    c: np.array, (nT, )
```

```
      spike counts

    s: np.array, (Dx * Dy, nT)
      stimulus matrix


    Returns
    -------

    f: float
      function value of the negative log likelihood at w

    """

    # ------------------------------------------------
    # Implement the negative log-likelihood of the LNP
    # ------------------------------------------------

    # define most important bits:
    w = w.reshape(-1, 1) # column vector
    wTs = w.T @ s
    r = np.exp(wTs) * dt * R

    # Compute negative log-likelihood
    f = np.sum(-c * np.log(r) + np.log(factorial(c)) + r) + (10 * np.sum(w)**2)

    # insert your code here
    return f


def deriv_negloglike_lnp_L1(
    w: np.array, c: np.array, s: np.array, dt: float = 0.1, R: float = 50
) -> np.array:
    """Implements the gradient of the negative log-likelihood of the LNP model

    Parameters
    ----------

    see negloglike_lnp

    Returns
    -------

    df: np.array, (Dx * Dy, )
      gradient of the negative log likelihood with respect to w

    """

    # ------------------------------------------------------------
    # Implement the gradient with respect to the receptive field `w`
    # ------------------------------------------------------------

    w = w.reshape(-1, 1) # column vector
    wTs = w.T @ s
    r = np.exp(wTs) * dt * R

    #df = -np.sum(s * (-c + r))

    df = ((-c + r) @ s.T) + (10* np.sign(w)).T
```

```
    return df
```

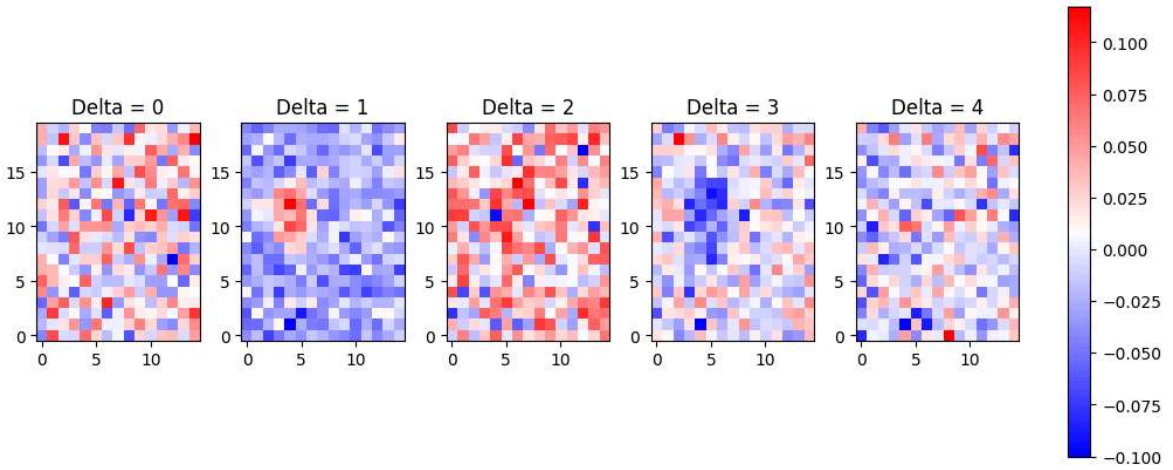
In [ ]:

```
from sklearn import linear_model

# insert your code here

# ----------------------------------------
# Fit the receptive field with time lags of
# 0 to 4 frames separately (the same as before)
# with sklern or pyglmnet (1 pt)
# ----------------------------------------

delta = [0, 1, 2, 3, 4]

# fit for each delay

Dx = 20
Dy = 15
dt = 0.1
R = 50

print(binned_spike_counts.shape)
print(s.shape)

# init w_hat
w_hat_L1 = np.zeros((Dx * Dy, len(delta)))

# Fit the receptive field for each time lag
for i, lag in enumerate(delta):

    if lag > 0:
        # remove last few stimulus frames according to lag (e.g. last 4 frames)
        # take the counts from "lag" onwards:
        # now, the "matching" stimulus will lag behind, and spike counts will be
        c_shifted = binned_spike_counts[lag:]
        s_shifted = s[:, :-lag]
    else:
        c_shifted = binned_spike_counts
        s_shifted = s
    rng = np.random.default_rng(5678)

    # Initial guess for the receptive field
    w_init = rng.normal(0, 1, Dx *Dy)

    # Perform the optimization

    result = minimize(
        fun=negloglike_lnp_L1,
        x0=w_init,
        args=(c_shifted, s_shifted, dt, R),
        jac=deriv_negloglike_lnp_L1,
        method='L-BFGS-B',
        tol=0.001
    )

    # Store the estimated receptive field
    w_hat_L1[:, i] = result.x
```

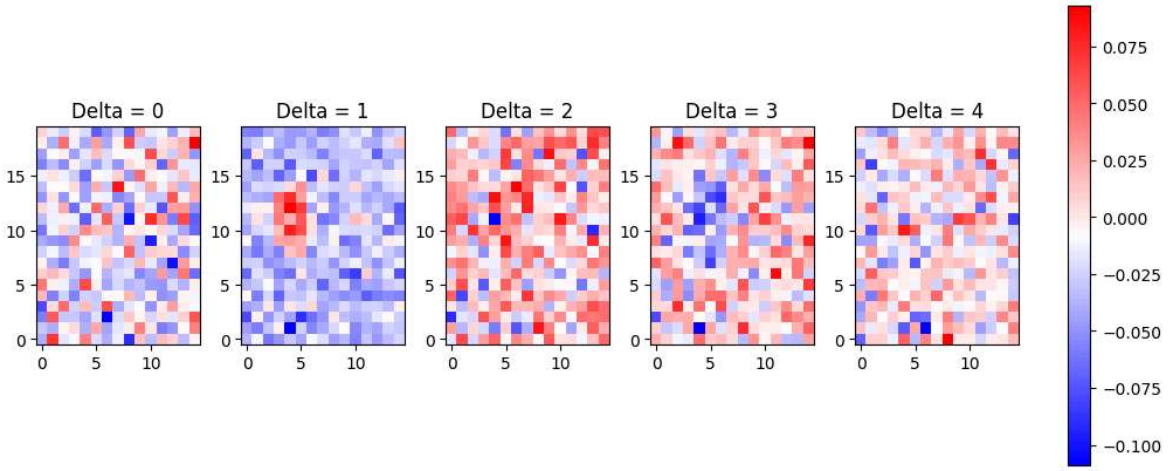
```
# did it work?
print("Optimization success:", result.success)
```

```
(1488,)
(300, 1488)
Optimization success: True
```

In [ ]:
```
# ----------------------------------------
# plot the estimated receptive fields (1 pt)
# ----------------------------------------

fig, ax = plt.subplot_mosaic(mosaic=[delta], figsize=(10, 4), constrained_layout
# add plot

for lag in delta:

    ax[lag].set_title(f"Delta = {lag}")
    w_reshaped = w_hat[:,lag].reshape(Dx,Dy)
    rf = ax[lag].imshow(w_reshaped, cmap='bwr', origin='lower')

cbar = fig.colorbar(rf, ax=[ax[0], ax[1], ax[2], ax[3], ax[4]],  orientation='ve

plt.show()
```



In [ ]:
```
fig, ax = plt.subplot_mosaic(mosaic=[delta], figsize=(10, 4), constrained_layout
# add plot

#
for lag in delta:

    ax[lag].set_title(f"Delta = {lag}")
    w_reshaped = w_hat_L1[:,lag].reshape(Dx,Dy)
    rf = ax[lag].imshow(w_reshaped, cmap='bwr', origin='lower')

cbar = fig.colorbar(rf, ax=[ax[0], ax[1], ax[2], ax[3], ax[4]],  orientation='ve

plt.show()
```


Delta = 0
Delta = 1
Delta = 2
Delta = 3
Delta = 4
0
5
10
15
0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.100