Neural Data Science

Lecturer: Prof. Dr. Philipp Berens

Tutors: Jonas Beck, Ziwei Huang, Rita González Márquez

Summer term 2022

Name: Marina Dittschar & Clarissa Auckenthaler

Coding Lab 7

In [1]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
import scipy.optimize as opt
import scipy.io as io
import scipy as sp
import scipy
import random
from math import e
from numpy.random import poisson
import matplotlib.cm as cm

mpl.rc("savefig", dpi=72)

sns.set_style('whitegrid')
%matplotlib inline
```

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 `sampleLNP` 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 (`logLikLnp`). 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.

Calculations

Simplify the log likelihood analytically and derive the analytical solution for the gradient. (2 pts)

See also: How to use Latex in Jupyter notebook.

$$\begin{aligned}
L(w) &= \log \prod_t \frac{r_t^{c_t}}{c_t!} \exp(-r_t) \\
&= \sum_t \log \frac{r_t^{c_t}}{c_t!} \exp(-r_t) \\
&= \sum_t \log r_t^{c_t} - \log c_t! + \log \exp(-r_t) \\
&= \sum_t c_t \log r_t - \log c_t! - r_t
\end{aligned}$$

Because $c_t!$ does not depend on w, we can move it to an additive constant. Using $r_t = \exp(w^T s_t) dtR$ we obtain:

$$\begin{aligned}
L(w) &= \sum_t c_t (w^T s_t + dtR) - \exp(w^T s_t) dtR + const_1. \\
&= \sum_t c_t w^T s_t - \exp(w^T s_t) dtR + const_2.
\end{aligned}$$

Note that s_t denotes a vector and c_t a scalar, in slight abuse of notation.

For the gradient:

$$\begin{aligned}
dL(w)/dw &= \sum_t c_t s_t - s_t \exp(w^T s_t) dtR \\
&= \sum_t (c_t - \exp(w^T s_t) dtR) s_t
\end{aligned}$$

This is interesting and makes intuitive sense: for the gradient, each stimulus frame is weighted by the difference between the observed and predicted count.

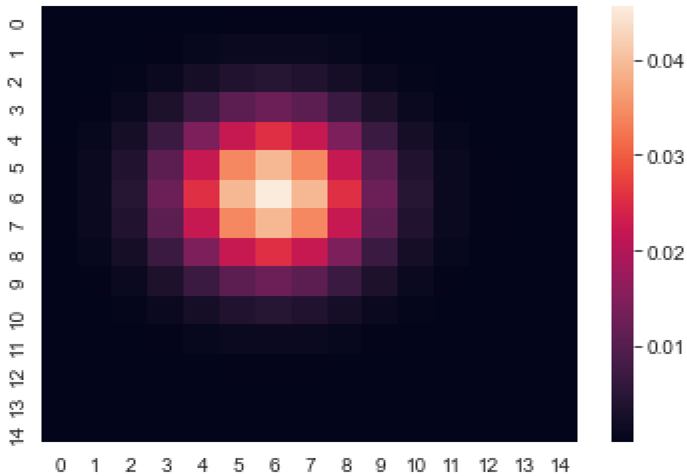
Generate data

In [2]:

```
def gen_gauss_rf(D, width, center=(0,0)):

    sz = (D-1)/2
    x, y = sp.mgrid[-sz: sz + 1, -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
```

```python
w = gen_gauss_rf(15, 7, (1,1))
sns.heatmap(w)
w.mean()
```

Out[2]: 0.0044444444444444444



In [3]:

```python
def sample_lnp(w, nT, dt, R, v):
    '''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 stimulus ensemble


    Returns
    -------

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

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

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

    Note
    ----

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

```
    '''

    np.random.seed(10)
    random.seed(10)

    # insert your code here

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

    s = [[random.randint(0,1) for i in range(w.shape[0])] for i in range(nT)]
    s = np.array(s)*np.random.normal(1, scale=v*4, size=nT)[...,np.newaxis]
    r = (e**(w.T*s)*R*dt).mean(axis=1)
    c = poisson(lam=r)

    return c, r, s
```

In [4]:

```
D = 15      # number of pixels in one dimension,
            # the simulated RF here is a square
nT = 1000   # number of time bins
dt = 0.1    # frame rate, 0.1s per bin.
R = 50      # firing rate in Hz
v = 5       # stimulus variance

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

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

Plot the responses of the cell.

In [5]:

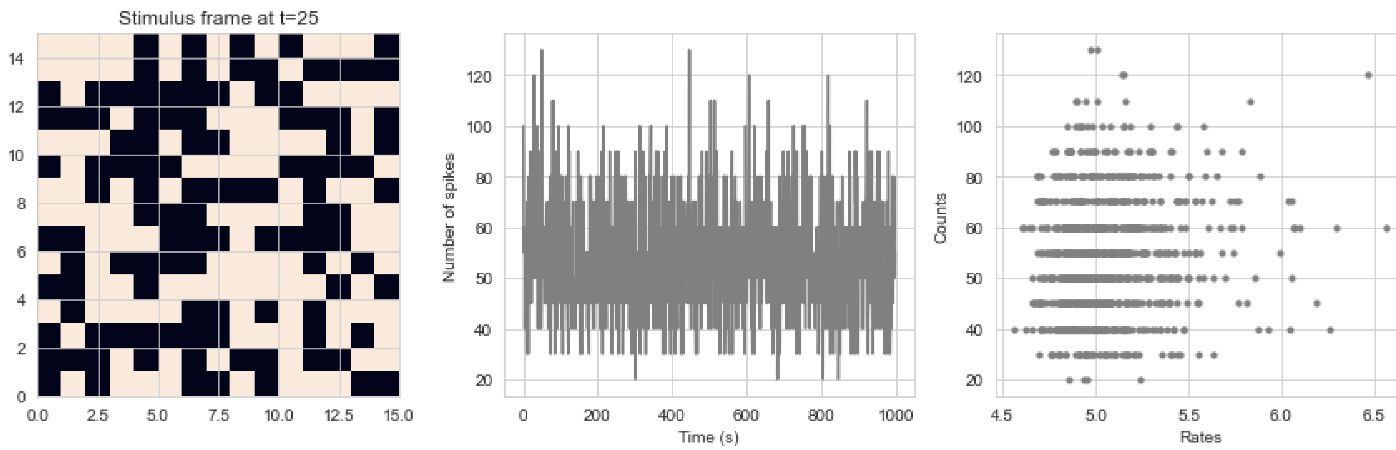
```
# insert your code here
# --------------
# Plot (0.5 pts)
# --------------
fig, axs = plt.subplots(1, 3, figsize=(15, 4))
# ------------------------------------------------
# (1) one example frame from s;
# ------------------------------------------------
s_im = s[25,:].reshape((15,15))
s_im= np.flipud(s_im)
axs[0].imshow(s_im, vmin = 0, vmax=1, extent=[0,15,0,15])
axs[0].set_title('Stimulus frame at t=25')
# ------------------------------------------------
# (2) the simulated response c;
# ------------------------------------------------
axs[1].plot(c,c='gray')
axs[1].set_xlabel('Time (s)')
axs[1].set_ylabel('Number of spikes')
axs[1].set_yticklabels([0,20,40,50,60,80,100,120,140])
# ------------------------------------------------
# (3) a scatter plot of r and c;
# ------------------------------------------------
axs[2].scatter(r,c,s=10,c='gray')
axs[2].set_xlabel('Rates')
axs[2].set_yticklabels([0,20,40,50,60,80,100,120,140])
axs[2].set_ylabel('Counts')
```

```
<ipython-input-5-6d21ac238425>:19: UserWarning: FixedFormatter should only be used togethe
r with FixedLocator
```

```
  axs[1].set_yticklabels([0,20,40,50,60,80,100,120,140])
<ipython-input-5-6d21ac238425>:25: UserWarning: FixedFormatter should only be used togethe
r with FixedLocator
  axs[2].set_yticklabels([0,20,40,50,60,80,100,120,140])
```

Out[5]:

```
Text(0, 0.5, 'Counts')
```



Implementation

Before you run your optimizer, make sure the gradient is correct. The helper function `check_grad` in `scipy.optimize` can help you do that. This 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.

$$
\begin{aligned}
L(w) &= \log \prod_t \frac{r_t^{c_t}}{c_t!}\exp(-r_t) \\
&= \sum_t \log \frac{r_t^{c_t}}{c_t!}\exp(-r_t) \\
&= \sum_t \log r_t^{c_t} - \log c_t! + \log \exp(-r_t) \\
&= \sum_t c_t \log r_t - \log c_t! - r_t
\end{aligned}
$$

In [6]:

```python
def likelihood(x, s, c):
    lik = 0
    for i in np.arange(s.shape[0]):
        temp = c[i]*np.log(e**(w.T*s[i,:])*R*dt)- np.log(scipy.special.factorial(c[i])) -
        print(temp)
        lik = lik + temp
    return lik



def gradient(x, s, c, dt=.1, R=50):
    grad = 0
    for i in np.arange(s.shape[0]):
        grad = grad + (c[i] - e**(x.T*s[i,:])*R*dt)*s[i,:]
    return grad

grads = gradient(w, s,c)
```

In [7]:

```python
def negloglike_lnp(x, c, s, dt=0.1, R=50):
```

```
    '''Implements the negative (!) log-likelihood of the LNP model and its
    gradient with respect to the receptive field w.

    Parameters
    ----------

    x: 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 x

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

    # insert your code here

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

    np.random.seed(30)
    r = np.exp(s@x.T)*R*dt

    f = np.sum(-c @ np.log(r)+ np.log(scipy.special.factorial(c))+ r)
    df = gradient(x, s,c)
    #print("Current f: ", f)
    return f, df
```

In [8]:

```
f = negloglike_lnp(x=w, c=c, s=s, dt=0.1, R=50)
```

Fit receptive field maximizing the log likelihood

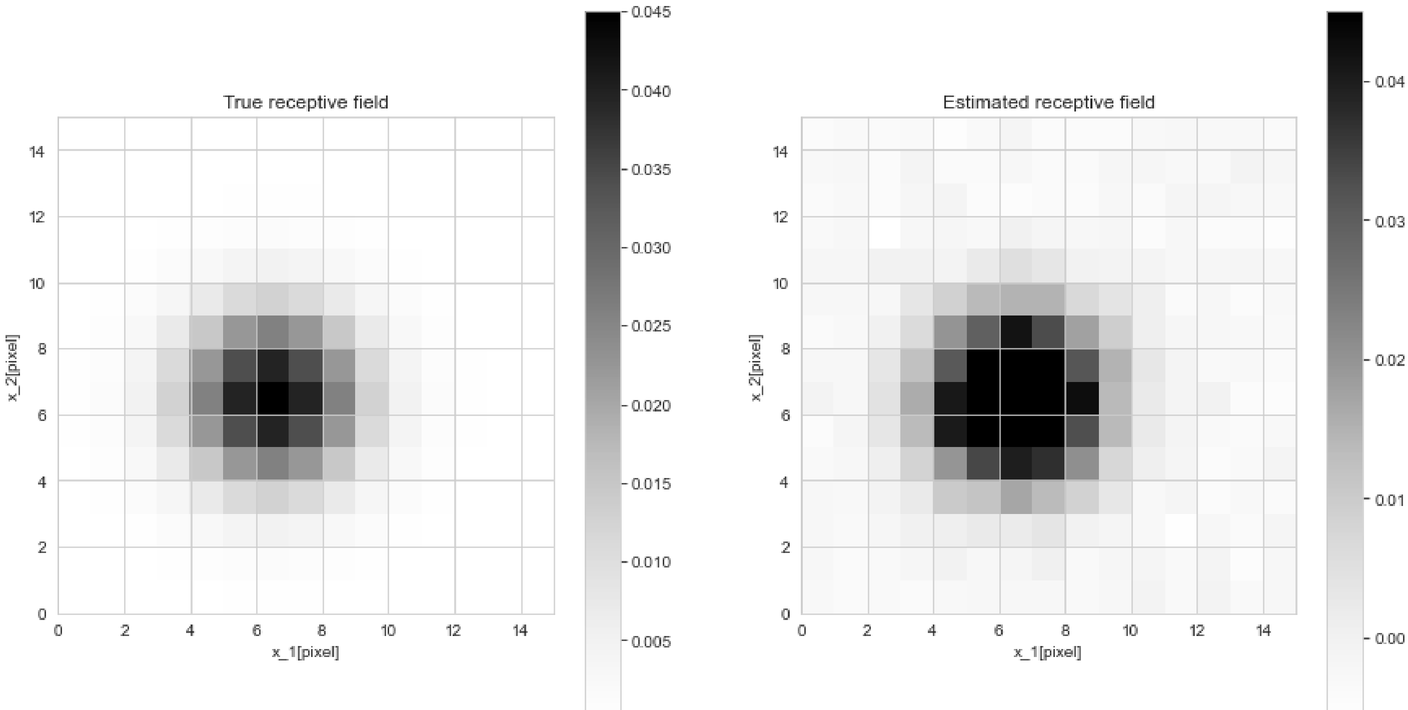
In [9]:

```
# insert your code here
# ------------------------------------------
# Estimate the receptive field by maximizing
# the log-likelihood (or more commonly,
# minimizing the negative log-likelihood).
#
# Tips: use scipy.optimize.minimize(). (1 pt)
# ------------------------------------------
res = scipy.optimize.minimize(negloglike_lnp, x0=w, args=(c,s), jac=True)
```

In [10]:

```
# insert your code here

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

```
# ------------------------------------

fig, axs = plt.subplots(1,2, figsize= (15,8))
result_task1 = res.jac.reshape((15,15))
result_task1 = np.flipud(result_task1)
result_task1 = result_task1 /np.sum(result_task1)
w_im = np.reshape(w, (15,15))
w_im= np.flipud(w_im)
plot0=axs[0].imshow(w_im, vmin=np.min(w), vmax=0.045, cmap='Greys',extent=[0,15,0,15])
axs[0].set_title("True receptive field")
axs[0].set_xlabel("x_1[pixel]")
axs[0].set_ylabel("x_2[pixel]")
plt.colorbar(plot0, ax=axs[0])
plot1= axs[1].imshow(result_task1, vmin =np.min(result_task1), vmax=0.045,cmap='Greys',ext
axs[1].set_title("Estimated receptive field")
axs[1].set_xlabel("x_1[pixel]")
axs[1].set_ylabel("x_2[pixel]")
plt.colorbar(plot1)
plt.show()
```



Task 2: Apply to real neuron

Download the dataset for this task from Ilias (`nda_ex_6_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 [11]:
```
var = io.loadmat(r'data/nda_ex_6_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 [12]:

```
# insert your code here

# ------------------------------------------
# Bin the spike counts at the same temporal
# resolution as the stimulus (0.5 pts)
# ------------------------------------------
c, bins = np.histogram(t, bins=trigger)
```

Fit receptive field for each frame separately

In [13]:

```
# 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)
# ------------------------------------------
res = scipy.optimize.minimize(negloglike_lnp, x0=np.zeros(300), args=(c,s.T),  jac=True)
res1 = scipy.optimize.minimize(negloglike_lnp, x0=np.zeros(300), args=(c[1:],s[:,:-1].T),
res2 = scipy.optimize.minimize(negloglike_lnp, x0=np.zeros(300), args=(c[2:],s[:,:-2].T),
res3 = scipy.optimize.minimize(negloglike_lnp, x0=np.zeros(300), args=(c[3:],s[:,:-3].T),
res4 = scipy.optimize.minimize(negloglike_lnp, x0=np.zeros(300), args=(c[4:],s[:,:-4].T),
```

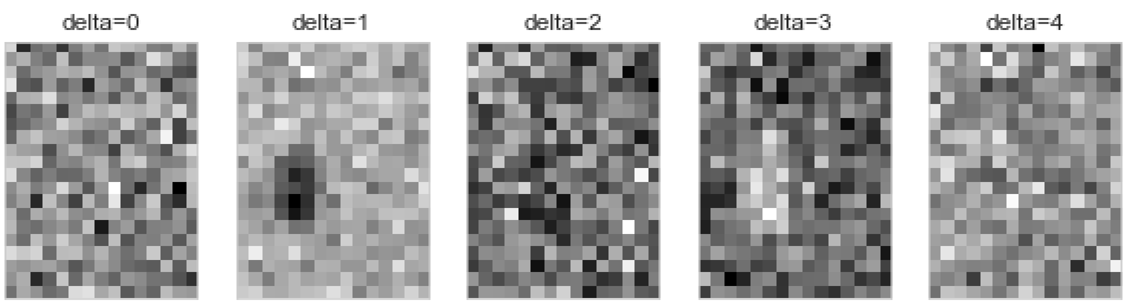
Plot the frames one by one

In [14]:

```
# insert your code here

# --------------------------------------------
# Plot all 5 frames of the fitted RFs (0.5 pt)
# --------------------------------------------
fig, axs = plt.subplots(1,5, figsize=(10,4))
result = res.jac.reshape((20,15))
result = result /np.sum(result)
result1 = res1.jac.reshape((20,15))
result1 = result1 /np.sum(result1)
result2 = res2.jac.reshape((20,15))
result2 = result2 /np.sum(result2)
result3 = res3.jac.reshape((20,15))
result3 = result3 /np.sum(result3)
result4 = res4.jac.reshape((20,15))
result4 = result4 /np.sum(result4)
w = np.reshape(w, (15,15))
axs[0].imshow(result, vmin=np.min(result), vmax=np.max(result),cmap=cm.gray)
axs[0].set_title("delta=0")
axs[0].set_xticks([])
axs[0].set_yticks([])
```

```
axs[1].imshow(result1, vmin=np.min(result1), vmax=np.max(result1),cmap=cm.gray)
axs[1].set_title("delta=1")
axs[1].set_xticks([])
axs[1].set_yticks([])


axs[2].imshow(result2, vmin = np.min(result2), vmax=np.max(result2),cmap=cm.gray)
axs[2].set_title("delta=2")
axs[2].set_xticks([])
axs[2].set_yticks([])

axs[3].imshow(result3, vmin = np.min(result3), vmax=np.max(result3),cmap=cm.gray)
axs[3].set_title("delta=3")
axs[3].set_xticks([])
axs[3].set_yticks([])

axs[4].imshow(result4, vmin = np.min(result4), vmax=np.max(result4),cmap=cm.gray)
axs[4].set_title("delta=4")
axs[4].set_xticks([])
axs[4].set_yticks([])
```

Out[14]: []



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 [15]:

```
W = np.empty((300,0))
for re in [res, res1, res2, res3, res4]:
    W = np.append(W,re.jac[...,np.newaxis], axis=1)
```

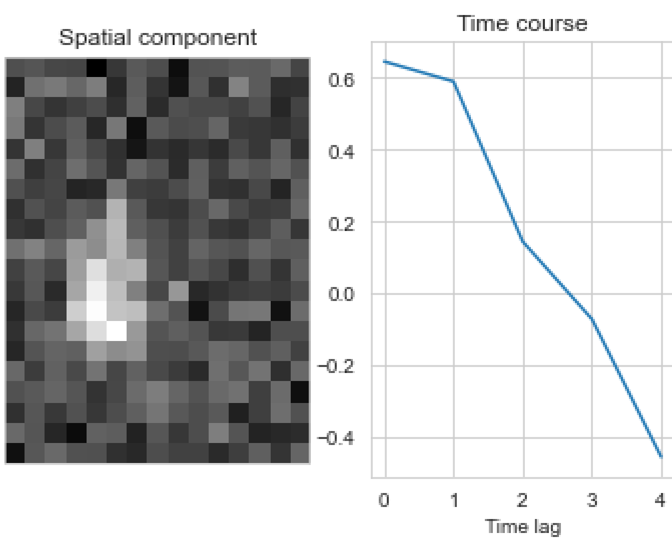
In [16]:

```
# insert your code here

# ------------------------------------------------
# Apply SVD to the fitted receptive field,
```

```
# you can use either numpy or sklearn (0.5 pt)
# -------------------------------------------
W = W - W.mean()
u,w,v= np.linalg.svd(W, full_matrices=False)
# -------------------------------------------
# Plot the spatial and temporal components (0.5 pt)
# -------------------------------------------
fig, ax = plt.subplots(1,2)
spatial = u[:,1]
spatial = np.reshape(spatial,(20,15))
ax[0].imshow(spatial, vmin=np.min(spatial), vmax=np.max(spatial),cmap=cm.gray)
ax[0].set_title("Spatial component")
ax[0].set_yticks([])
ax[0].set_xticks([])

ax[1].plot(v[:,1])
ax[1].set_title("Time course")
ax[1].set_xlabel("Time lag")
```

Out[16]:

```
Text(0.5, 0, 'Time lag')
```



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 [17]:

```
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)
# ------------------------------------------
lasso = linear_model.Lasso(alpha=0.03)
lasso.fit(s.T, c)
lasso1 = linear_model.Lasso(alpha=0.03)
lasso1.fit(s[:,:-1].T, c[1:])
lasso2 = linear_model.Lasso(alpha=0.03)
lasso2.fit(s[:,:-2].T, c[2:])
lasso3 = linear_model.Lasso(alpha=0.03)
lasso3.fit(s[:,:-3].T, c[3:])
lasso4 = linear_model.Lasso(alpha=0.03)
lasso4.fit(s[:,:-4].T, c[4:])
```

Out[17]:

```
Lasso(alpha=0.03)
```

In [18]:

```
# insert your code here

# -------------------------------------------------
# Plot all 5 frames of the fitted RFs, compare them
# with the ones without regularization (0.5 pt)
# -------------------------------------------------

res_sklearn = lasso.coef_.reshape((20,15))
res1_sklearn = lasso1.coef_.reshape((20,15))
res2_sklearn = lasso2.coef_.reshape((20,15))
res3_sklearn = lasso3.coef_.reshape((20,15))
res4_sklearn = lasso4.coef_.reshape((20,15))
fig, ax = plt.subplots(2,5, figsize=(15, 10))

ax[0,0].imshow(result, vmin=np.min(result), vmax=np.max(result))
ax[0,0].set_title("delta=0")
ax[0][0].set_xticks([])
ax[0][0].set_yticks([])

ax[0,1].imshow(result1, vmin=np.min(result1), vmax=np.max(result1))
ax[0,1].set_title("delta=1")
ax[0][1].set_xticks([])
ax[0][1].set_yticks([])

ax[0,2].imshow(result2, vmin = np.min(result2), vmax=np.max(result2))
ax[0,2].set_title("delta=2")
ax[0][2].set_xticks([])
ax[0][2].set_yticks([])

ax[0,3].imshow(result3, vmin = np.min(result3), vmax=np.max(result3))
ax[0,3].set_title("delta=3")
ax[0][3].set_xticks([])
ax[0][3].set_yticks([])

ax[0,4].imshow(result4, vmin = np.min(result4), vmax=np.max(result4))
ax[0,4].set_title("delta=4")
ax[0][4].set_xticks([])
ax[0][4].set_yticks([])
```

```
res_sklearn_all = np.array([res_sklearn, res1_sklearn, res2_sklearn, res3_sklearn, res4_sk

ax[1,0].imshow(res_sklearn, vmin=np.min(res_sklearn_all), vmax=np.max(res_sklearn_all))
ax[1,0].set_yticks([])
ax[1,0].set_xticks([])


ax[1,1].imshow(res1_sklearn, vmin=np.min(res_sklearn_all), vmax=np.max(res_sklearn_all))
ax[1,1].set_yticks([])
ax[1,1].set_xticks([])

ax[1,2].imshow(res2_sklearn, vmin=np.min(res_sklearn_all), vmax=np.max(res_sklearn_all))
ax[1,2].set_yticks([])
ax[1,2].set_xticks([])

ax[1,3].imshow(res3_sklearn, vmin=np.min(res_sklearn_all), vmax=np.max(res_sklearn_all))
ax[1,3].set_yticks([])
ax[1,3].set_xticks([])

ax[1,4].imshow(res4_sklearn, vmin=np.min(res_sklearn_all), vmax=np.max(res_sklearn_all))
ax[1,4].set_yticks([])
ax[1,4].set_xticks([])
plt.text(-72, -5, "alpha =0.03")
```

Out[18]:

```
Text(-72, -5, 'alpha =0.03')
```