Informatics 1 Cognitive Science - Computational Cognition: Models of Categorisation

---

1. Similarity Based Categorisation

2. Simple Neuron Models

3. Visual Neuron Receptive Field Predictive Modelling

4. Memory Based Cognition

In [64]:
```
# imports required for all tasks

# import numpy, the python scientific package
import numpy as np

# test images from scipy
from scipy import misc

# import maplotlib to plot the visualizations
from matplotlib import pyplot as plt
#this is to output the plots in the notebook
%matplotlib inline

# Let's also hide unnecessary warnings
import warnings
warnings.filterwarnings('ignore')
```

Part 1

Similarity-based models of categorization.

In [65]:
```
# Here we use the following functions:

def similarity(x,y,c):
    return np.exp(-c*np.sum(np.power((x-y),2)))

def similarityToAll(x,yM,c):
    xM=np.tile(x,[np.size(yM,axis=0),1])
    sqDiff = np.power(xM-yM,2)
    ssDiff=np.sum(sqDiff,axis=1)
```

```
    return np.exp(-c*ssDiff)

def model1(testCase,cat1,cat2,c):
    c1s = np.sum(similarityToAll(testCase,cat1,c))
    c2s = np.sum(similarityToAll(testCase,cat2,c))
    return c1s/(c1s+c2s)

def model2(testCase,cat1,cat2,c):
    c1s = similarity(testCase,np.average(cat1,axis=0),c)
    c2s = similarity(testCase,np.average(cat2,axis=0),c)
    return c1s/(c1s+c2s)
```

Parameterisation

First, let's understand what c does -- it's a parameter we can adjust to change the way our similiarity function behaves. Suppose we have three examples, x1=[1, 1], x2=[1,0], and x3=[1,3].

In [66]:
```
for i in range(1):
     #x1
    print("Similarity in range 0-5: ", similarity(1,1,i))
```

```
Similarity in range 0-5:  1.0
```

In [67]:
```
print("Similarity for (1,0): ")
for i in range(0,5):
     #x2
    print(similarity(1,0,i))
```

```
Similarity for (1,0):
1.0
0.36787944117144233
0.1353352832366127
0.049787068367863944
0.01831563888873418
```

In [68]:
```
print("Similarity for (1,3): ")
for i in range(0,5):
     #x3
    print(similarity(1,3,i))
```

```
Similarity for (1,3):
1.0
0.01831563888873418
0.00033546262790251185
6.14421235332821e-06
1.1253517471925912e-07
```

1. What happens to the relative similarity of x1 to x2 and x3 as we let c become very large?

The greater the c value, the similarity of x1 to x2 and x3 diminishes as the increasing c values make x2 and x3 approach 0.0, whereas x1 consistently returns 1.0. Because x1 is always returning the similarity between 1 and 1, which will always be 100% or 1.0. The x2 and x3 functions not only decrease similarity to x1 because they compare two different values, but also because the function shows the similarity infinately approaches 0.0 the higher the c value. Between x2 and x3 they compare different values, the x3 function ends up much more rapidly approaching 0.0 as c values increase, wheras the x2 function also, but much more slowly, approaches 0.0 as c values increase.

2. What happens if we let c become very close to zero?

The closer the c value gets to zero, all functions infinately approach 1.0 since the for the similarity function the limit of similarity(x) as x->infinity+ = 1.0. Since as x approaches positive infinity, the values cannot equal zero according to the similarity() function's asympotes, vertically at 0.0 since there is a discontinuous y-intercept where if x=0.0 then (0.0, 1.0).

3. Why shouldn't we let c be negative?

If the c value is marked as negative, x2 and x3 cannot function properly because it returns exponential values in the wrong quadrant, which doesn't properly weigh the similarity.

Model Comparison

Second, let's look at some simple models. Two functions in the block above, model1 and model2, describe minimal versions of categorization models based on similarity.

In [69]:
```
def model1(testCase,cat1,cat2,c):
    c1s = np.sum(similarityToAll(testCase,cat1,c))  #similarityToAll used to
    c2s = np.sum(similarityToAll(testCase,cat2,c))
    return c1s/(c1s+c2s)

def model2(testCase,cat1,cat2,c):
    c1s = similarity(testCase,np.average(cat1,axis=0),c)
    c2s = similarity(testCase,np.average(cat2,axis=0),c)
    return c1s/(c1s+c2s)
```

In [70]:
```
def similarityToAll(x,yM,c):
    xM=np.tile(x,[np.size(yM,axis=0),1])
    sqDiff = np.power(xM-yM,2)
    ssDiff=np.sum(sqDiff,axis=1)
    return np.exp(-c*ssDiff)
```

4. Which categorization model does model1 best correspond to? Explain.

Both categorization models organize concepts based on their overall similarity. Model 1 best corresponds to Rosch's Exemplar Theory because it finds a single representation of the total similarity to exemplar cases by using the similarityToAll function summed with the total resemblance to non-exemplar cases. Which highlights the most similar resemblance to a catgory and returns the similarity to the first.

5. Which categorization model does model2 best correspond to? Explain.

Although this model is also a Similarity-based theory it is slightly more juvenile in execution of the Prototype catgeorization theory, because it compares the testCase concept to the averaged prototype, found by comparing and returning the average similarity of the two values for each category.

Third, let's consider the problem of categorizing a new creature, named Bob, as a wimble or as a womble. The only features our creatures can have are volume and mass; a creature with a volume of $1cm^3$ and a mass of $2g$ can be described as [1 2].

The next block describes previously-observed wimbles and wombles in terms of these features, as well a new creature, which we will call Bob. For the rest of these questions, assume $c = 0.1$.

In [71]:
```
wimbles = np.matrix([[1,1],[1,1],[1,1],[1,1],[1,10],[1,10],[1,10],[1,10]])
wombles = np.matrix([[1,5.01],[1,5.02],[1,5.03],[1,5.04],[1,5.05],[1,5.06],[

bob = np.array([1,5.5])
```

Classification

Lets classify these models into two arbitrary categories: Wimbles and Wombles.

Suppose someone is playing a simple video game, and so far has only encountered the wimbles and wombles listed above. They are asked to make a judgement about what Bob is, without being given any guidance or restrictions about Bob's category or knowing what categories exist in the game.

In [72]:
```
c=0.1
model1(bob,wimbles,wombles, c)
```

```
Out[72]: np.float64(0.11875771662509463)

In [73]: model2(bob,wimbles,wombles, c)

Out[73]: np.float64(0.5051754401546502)

In [74]: wimbles = np.matrix([[1,1],[1,1],[1,1],[1,1],[1,10],[1,10],[1,10],[1,10],[1,

In [75]: model1(bob,wimbles,wombles, c)

Out[75]: np.float64(0.5740361569327478)

In [76]: model2(bob,wimbles,wombles, c)

Out[76]: np.float64(0.5051754401546502)

In [77]: wimbles = np.matrix([[1,1],[1,1],[1,1],[1,1],[1,10],[1,10],[1,10],[1,10]])

         wombles = np.matrix([[1,5.01],[1,5.02],[1,5.03],[1,5.04],[1,5.05],[1,5.06],[

In [78]: model1(bob,wimbles,wombles, c)

Out[78]: np.float64(0.0147526245153882)

In [79]: model2(bob,wimbles,wombles, c)

Out[79]: np.float64(0.5051754401546502)
```

6. What does Model 1 classify Bob as?

Model 1 classifys Bob as most similar to a womble because the value returned is very unlikely to be a part of the wimble category(c1s), returning a low value indicating a low resemblance to wimbles. Since the entire model is reshaped by the similarityToAll function taking the power of the difference of the larger repeating arrays xM and yM and as output returns the likelihood the case is the first category, wimbles. Model1 returning 0.1188, shows it has little resemblance to wimbles. If the first category was switched to wombles, the model returns the decimal percent of resemblance to wombles exactly 1.0-0.1188=0.8812, or 88.12% resemblance to womble.

7. What does Model 2 classify Bob as?

Model 2 classifys Bob as a wimble since the model analysed the similarity as the sum of the power of the difference between each x and y(Bob and the prototype). The model compares Bob's similarity to the averaged prototype comparing similarity to the first category, wimbles. Each prototype is the average of each category's,wimbles and wombles' respective combined value. So the value returned is always the similarity to

the wimble prototype average and always returns 0.50517, more than 50% resemblance to the wimble prototype.

8. Suppose we next see 9 more groups of wimbles, all identical to the group above. What changes in the models' predictions?

When adding 9 more groups of wimbles, model 1's prediction gets larger because it takes a sum of each category's similarity to each exemplar, and with far more wimbles now than wombles, the value returned slightly grows indicating little but a higher resemblance of 0.5740 to the wimble's exemplar. Because before the addition of so many wimbles, model 1 returned only 0.1188 resemblence to a wimble, this addition makes it so the model is more ambiguous about how little resemblance Bob has to the wimbles' category exemplar based on the comparatice amount of members of each category. Model 2 does not change at all however, since the model itself is not a sum and instead averages the values of each catgegory, wimbles and wombles, then sums and averages Bob's similarity to a wimble. Which is why it always returns the same similarity.

9. Is it *reasonable* for the predictions to change if we see many more examples of one category than another?

It is reasonable to expect some change when introducing an abundance of members to one category, since the distribution of members per each category becomes heavily weighted. For example if the opposite category of wombles added 9 more groups of members, model 1 inclines more heavily toward Bob being a womble (see Out[15]:) and the addition of so many more wombles makes it so the resemblance to a wimble is distinguished. Yet this is not always the case depending if combined consideration of all values is contingent to each individual model. The first model proved that categories' amount of members is contingient to the model and most likely loses some accuracy in the process, since the values similarity changes according to uneven distribution of number of members in a category to an abundance of members in another. And the second model showed it's analysis of resembance to a category's prototype is not going to change and it would not be reasonable to expect it to since the prototype does not become skewed with additional members.

Limitations

Finally, let's consider some of the shortcomings of these models.

10. What is one way both models above fail to account for human category judgments in this scenario, in terms of the

kinds of information people consider, the judgments they make, and/or how they represent categories? Where possible, discuss specifics of the current example.

When looking to make a judgement decision about what category Bob falls under humans decide based on the concepts they've been exposed to. Although both models fail to account for human unpredictability, humans can make irrational categorizations and judgements even when the consequence outweighs personal gain-- take for example, the prisoners dilemma where the rational assumption is that any prisoner would want to screw over the other prisoner, but when given the opportunity to repeat the instance again and again, people end up considering more information than quanitified external features, such as age, time and location, and optimal personal outcome, etc. These two models rely on information about the categories and their respective arrays to compare similarity to some manipulation of the catgorical info. The models lack accountability for irrational associations, commonly caused by emotions and situational consequences of a concept between categories. Those circumstances can randomly affect human judgement, a failure to model a prototype or exemplar by operating off of categorical data alone. Neither model allows for randomly or unpredictably chosen categorization even when it may not make sense!

11. In addition to the first failure, name a second one that a model in the Sanborn et al. paper fixes. That is, a problem faced by Model 1 or Model 2 above, but not (either) the MAP model or the particle filter in the Sanborn paper discussed in lecture.

The particle filter model has the highest quality of categorization and can find the posterior probability of the extremes within a category, using a framework of the maximum likely extreme and judging which category is best fit for the concept. The order of concepts and the way that they're mental representations are stored and remembered affects human decision making and predictions, and the MAP model learns from data as it is presented in the order it is presented with to produce lower quality categorization than the particle filter which also uses sequential data and allows it to influence the dataset in order for categorization.

Part 2

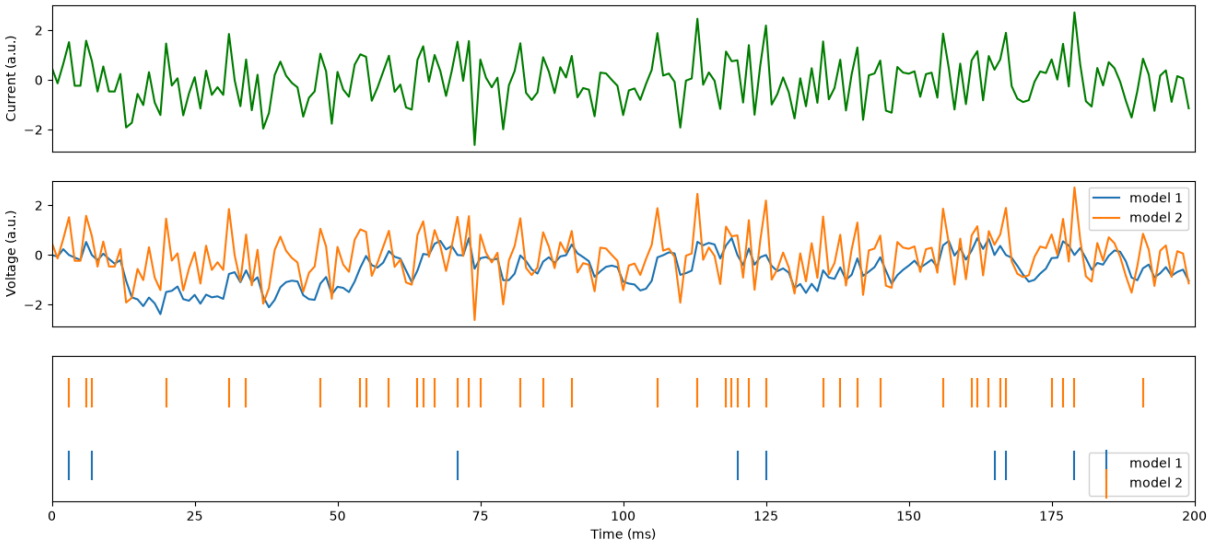
Simple neuron models.

The code below simulates two neuron models. Each neuron receives the same input current $I(t)$, which here is random and drawn from a Gaussian distribution.

In [80]:
```
# Simulation parameters
```

```
# simulation time (in milliseconds)
t_max = 200

# input current
np.random.seed(42) # fix random seed
I = np.random.normal(size=t_max)

# Spike threshold
threshold = 0.75

# Plot the input current
plt.figure(figsize=(16,7))
plt.subplot(311)
plt.plot(I, 'g')
plt.xlim(0,t_max)
plt.xticks(())
plt.ylabel('Current (a.u.)')

# Voltage and spikes for model 1
V1 = np.zeros(t_max)
spikes1 = []
for t in range(1,t_max):
    V1[t] = V1[t-1] + 1/10 * (-V1[t-1] + 4.46*I[t])
    # aside: the factor 4.46 in the line above was added
    # to make V1 and V2 have approx. the same variance (when there are no sp
    if V1[t]>threshold:
        spikes1.append(t)
        V1[t] = 0


# Compute voltage and spikes for model 2
V2 = I
spikes2 = np.where(V2>threshold)[0]

# Neuron activation (voltage) for both models
plt.subplot(312)
plt.plot(V1, label='model 1')
plt.plot(V2, label='model 2')
plt.xlim(0,t_max)
plt.xticks(())
plt.ylabel('Voltage (a.u.)')
plt.legend()

# Spike trains generated by both models
plt.subplot(313)
plt.scatter(spikes1, np.zeros(len(spikes1)), marker='|', s=500, label='model
plt.scatter(spikes2, np.ones(len(spikes2)), marker='|', s=500, label='model
plt.xlim(0,t_max)
plt.ylim(-.5,1.5)
plt.yticks(())
plt.xlabel('Time (ms)')
plt.legend()
np.std(V1), np.std(V2)
```

Out[80]: (np.float64(0.6713385585715156), np.float64(0.9286734887354714))



Analysis

Lets compare these two Models and explain how these models work and state the equations.

These two models operate slightly differently with Model 1's equation is V1= v[t-1] + 0.1 * (-V1[t-1] + 4.46* I[t]) and Model 2's equation is the input current: np.random.normal(size=t_max) through the time threshold 200ms and each model activates each time the voltage surpasses the threshold, 0.75. The first model computes the voltage of the frequency 0.1 to the change in voltage (ΔV) in terms of the current input's variance. ΔV is subtraction of the previous value from the input current or (-V1[t-1] + 4.46* I[t]). While Model 1 activates then returns to resting state of 0 and the time(ms) it fired is marked by a tick before the next opportunity to activate. The second model however uses V2 or I, the randomly generated current over time, that changes with respect to the input current and array values When the voltage/time of Model 1 surpasses the threshold it activates then returns to a resting state of 0 and the time(ms) of firing is also marked by a tick. Although this function does not have any restictions to return to a baseline resting state and will continue to grow and lose current acording to the input current, activating every time surpassing 0.75, the threshold.

Comparison

The spiking differs between the two models, lets explain which components of the model cause those differences.

The two models have a very different spiking arrangment since the input current Model 2 has much more frequent spikes than Model 1 which has infrequent activation clusters and moments. Model 1 spikes by every activation per millisecond in 200ms(its range) + 0.10 * (V1[t-1] + 4.46* I[t]). Model 1 also resets after each activation by reaching the threshold, as shown through the 'if' statement for V1, both referring to itself and resetting the data after each activation makes Model 1 more filtered and efficent with voltage expulsion. Model 2 spikes by the np.where function that indexes the time of each activation of I(t)'s random normal function and does not have any form of resetting since it outputs the input current, 'I'. The component that causes the differences is largely due to how Model 1 has a short term memory of data almost, by resetting all or nothing

Efficacy of Neuronal Models

Lets explore which aspects of the activity of real neurons are captured by each model, and which are missing.

The construction of model one has a condition if V1[t]>threshold: / spikes1.append(t) /V1[t] = 0 that mimics the firing action potential of neural messages in response to a world of input. One quality shown here is first that the model has threshold for spiking, forming the all-or-nothing action potential of neurons since they either fire or do not reach the threshold and do not fire. Also this model outlines the neural resetting after any spike/activation, where our neurons return to a homeostasis after firing and passing off the message. Unfortunately this model is missing the neural memory that we store in our brain, which could have been mimiced within the V1 dataset to allow an indexed value to be stored and add to the remembered voltage, which would allow for more complex neural cells to be simultaneously modelled. The model also does not have any data for the individual associative memory we allocate to every firing pattern(thought pattern's that are stored in our Long term memory), these additonal pieces of 'what' and 'where' information contributed to the future action potential of any one neuron, and how closely/or differently resembling one category or dataset affects the need to fire again, since our semantic memories are connected to our categorial judgements.

Part 3

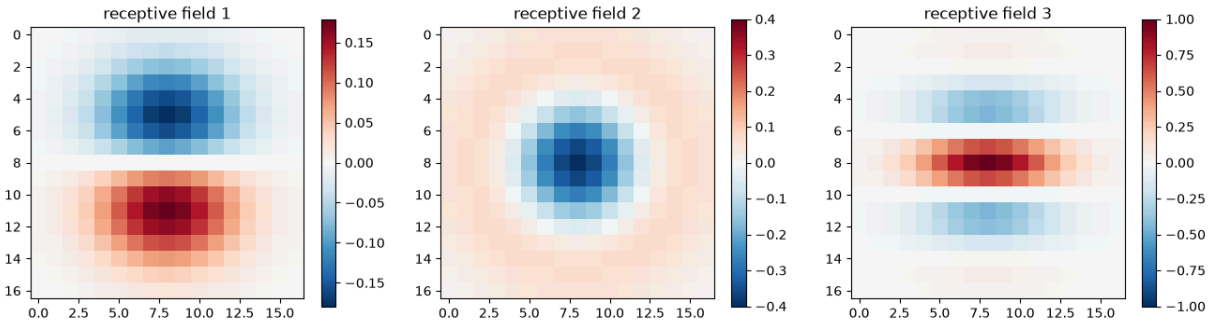
Receptive fields in the visual system.

In this exercise we will explore receptive field models of neurons in the visual system. Specifically, here we will look at the spatial filtering neurons perform, and ignore the temporal aspects. Then we can define the receptive field of a neuron as a matrix of

weights that scale each pixel, before the output is computed by summing the weighted pixels.

The following code predictively creates three different receptive fields which model the visual system's inputs.

In [81]:
```
x = np.tile(np.arange(-8,9),(17,1))
y = x.T
s1=2
s2=4

# the first kernel
s1=3
kernel1 = np.exp(-(x*x+y*y)/(2*s1**2))*np.sin(y/10)


# the second kernel
kernel2 = -np.exp(-(x*x+y*y)/(2*s1**2)) + 0.6*np.exp(-(x*x+y*y)/(2*s2**2))


# the third kernel
s1=3
kernel3 = np.exp(-(x*x+y*y)/(2*s1**2))*np.cos((x.T/1.25+np.cos(np.pi/2)*y[:,

# plot the kernels
plt.figure(figsize=(16,4))
for i,kernel in enumerate((kernel1,kernel2,kernel3)):
    plt.subplot(1,3,i+1)
    maxv = np.max((np.abs(np.min(kernel)), np.max(kernel)))
    plt.imshow(kernel, cmap=plt.cm.RdBu_r, vmin=-maxv, vmax=maxv)
    plt.colorbar()
    plt.title('receptive field {}'.format(i+1))
```



Analysis

Of the three receptive fields shown above, lets explain how each cell type best matches to each receptive field, and where in the visual these cells are found.

Which feature in an image most strongly excites each cell type, and which features yield the weakest response?

The receptive fields above are each produced by different instances of different retinal neural cells, and their specific responses to different features in the visual field. Each of the receptive fields are produced by an individual kernel that resembles a retinal ganglion cell in the visual pathway. Receptive field 1 is made of a negative semicircle above a positive semicircle of values composed of the sine function multiplied by the Gaussian model exponential function e^((x* x+y* y)/(2* s1** 2)). Receptive field 1 best resembles a V1 complex cell with edge detection features between two different variables, meaning it models more excitatory values nearer to the focused edge, a weaker response away from the edge, and the weakest response is from any object with an entirely different orientation. This cell type is especially sensitive to spacial orientation and more importantly positionally-invariant, meaning the cell responds to a preferred spacial orientation and adjustments to that orientation yield weaker responses as the orientation differs from the cells' predisposition.

Receptive Field 2 is made of a focused negative circle or elliptical with a slightly positive background. The function is built by a negative Gaussian exponential function and another Gaussian exponential function of a larger size. This best matches a Lateral Geniculate Nucleus(LGN) cell with an 'on'-center an 'off'-surround focus, meaning the cell models excited responses or strong negative values instead of weak negative values at the center focus and less excited responses or a weakened response to the background.

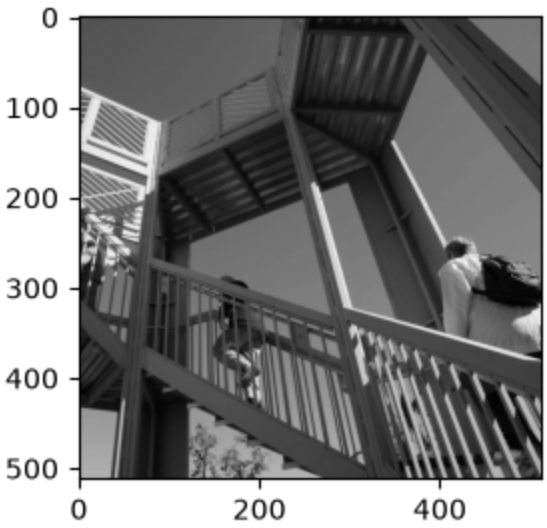
Receptive Field 3 is made up of alternating positve and negative horizontal stripes composed of the multiplication of a cosine function and the Gaussian exponential function producing not nearly as negative of values. Therefore the edges between the values are not as strong. This best matches a V1 simple cell that is able to detect some finer pattern or grains, meaning the cell is able to detect fine edges, and spacial orientation. The simple cell, similar to the V1 complex cell is sensitive to specific spacial orientation. This cell also has a predispositional higher activation around its' preferred direction and location of any object in the receptive field.

Natural Image Processing

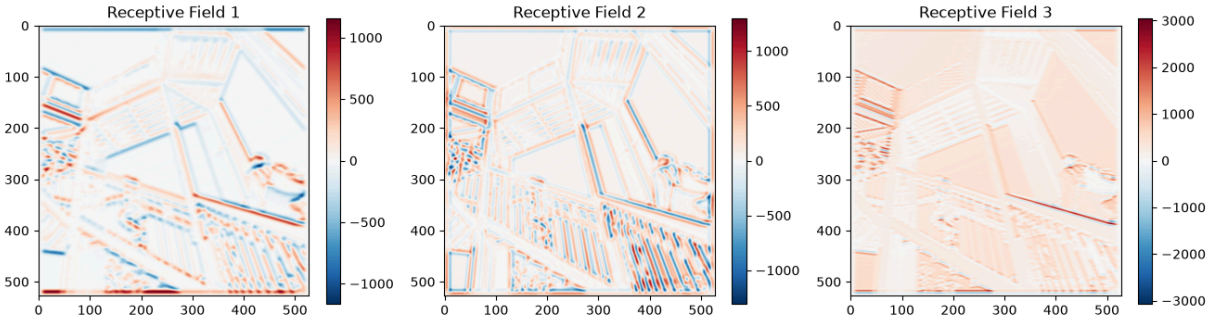
Now lets test how filtering these receptive fields perform by computing the responses of these neurons to a natural image. To this end we assume we have as many neurons as pixels in the image, and the centre locations of the receptive fields match the pixel locations. Then we can use a simple convolution operation to predict the response of each neuron.

Below are the predicted responses for each receptive field. The Blue-Red colour scale (using a diverging colormap with the zero clearly visible, as above), indicates positive and negative responses of the model.

In [86]:
```
# here we use the 2D convolution function in scipy
from scipy.signal import convolve2d
from scipy import datasets
# The test image as a 2D matrix:
image = datasets.ascent()

# to see the original image, use:
plt.figure(figsize=(16,3))
plt.subplot(1,4,1)
plt.imshow(datasets.ascent(),cmap=plt.cm.gray)

# the result for the first receptive field
conv = convolve2d(image,kernel1)
```



In [88]:
```
'''Convolve img for Kernel 2'''
conv2 = convolve2d(image,kernel2)
'''Convolve img for Kernel 3'''
conv3 = convolve2d(image,kernel3)

plt.figure(figsize=(16,4))
for i,convv in enumerate((conv,conv2,conv3)):
    plt.subplot(1,3,i+1)
    maxv = np.max((np.abs(np.min(convv)), np.max(convv)))
    plt.imshow(convv, cmap=plt.cm.RdBu_r, vmin=-maxv, vmax=maxv)
    plt.colorbar()
    plt.title('Receptive Field {}'.format(i+1))
```



Results

Let's describe and explain these results. The receptive fields are indeed selective to the image features you predicted above, similar to how neurons in the visual system respond in this way.

The results indicated each receptive field [1,2,3] with its respective kernel's [1,2,3] neural feature detection across the image, showing a gradient of postive and negative values to model the staircase as a given receptive field according to the visual features of the cells each kernel resembles. Although, receptive fields are usually a collection of many much smaller circular center-surround structured fields in a grid each for perception of oppositely responding values of positive center negative surround for the field, these three receptive fields model such processes and are indeed similar to the image features I predicted (in 3a) based on the kernel models.

The first kernel I predicted to model a complex V1 cell in receptive field 1 with some spatial orientation and edge detective features, with higher excitatory signals closer to the edge with a positive and its opposite negative edge, both with weaker signals gradually away from the edge. As shown by the model above(see 3b). These cells respond in this way because these neurons allow us to better respond to their preferred spacial orientation of an object in the field.

The second kernel I predicted to model a simple LGN cell that has negative focus and positive surrounding excitatory signals. These lateral geniculate cells in the visual system make up the 'what' and 'where' channels of information later sent to the occipital lobe for processing.

The third kernel I predicted to model a V1 simple cell with it's specific spacial orientation dependent excitation for the more fine grained details, as shown majority positively and ever so slightly negative values on opposing edges, with zero excitation in areas that does not meet the cell's preferred location or orientation. The orientation of the simple cell is most likely positionally-invariant to a diagonal line from top left to low right or similar to a descending line (ex. f(x)=-x), this is most likely the case because the cell has an extremely positive assertion on similar lines.

The neurons in the visual system respond to their individual purposes, such as edge detection features, selective spacial orientation, and center-surround structured reception features, which collectively compose our perception of any visual field before someone. They respond to their individual features, and such as a cell with selective spacial location and orientation, neural signals will only respond in the receptive field to their preferred orientation of a bar or edge.

Part 4

A memory model.

Here we simulate the Hopfield model. This model simulates M binary neurons with the following activity rule:

$$s_i(t+1) = \Theta\left(\sum_{j=1}^{M} w_{ij} s_j(t) - \theta_i\right)$$

$$\Theta(a) = \begin{cases} 1 & a \geq 0 \\ 0 & a < 0 \end{cases}$$

The variable $s_i(t)$ is the activity of neuron i at time t, and w_{ij} is a weight matrix connecting the neurons. In the following we set the bias $\theta_i = 0$.

The following learning rule is used to store a set of N binary patterns $p_i^n \in \{0, 1\}$ in this network:

$$w_{ij} = \frac{1}{N} \sum_{n=1}^{N} (p_i^n - s)(p_j^n - s)$$

The quantity s is the sparseness of the patterns, which is the fraction of elements in the patterns that are 1. So if a pattern of length 20 contains 10 times the '0' and 10 times '1', $s = 0.5$.

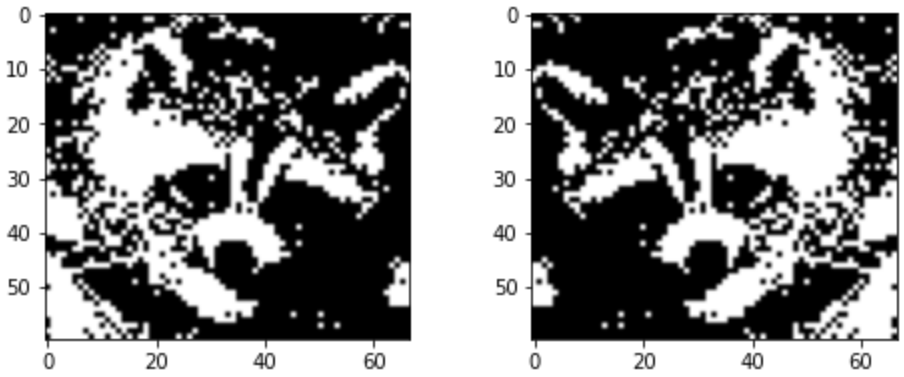
In other words, sparseness is the arithmentic mean (average) over all patterns: $s =< p^n >_N$.

An excellent summary of the Hopfield network is in chapter 42 in the following textbook (free online): *MacKay, D. J. C. (2003). Information Theory, Inference, and Learning Algorithms*

In [ ]:
```
# Let's create a binary pattern to store, here we use a racoon:

image = datasets.face(gray=True)[:600:10,280:950:10]>140
pattern = image.flatten().astype(int)
```

```
plt.figure(figsize=(8,3))
plt.subplot(121)
plt.imshow(image, cmap=plt.cm.gray)

# We also use the mirror image of the same racoon

image2 = image[:,::-1]
pattern2 = image2.flatten().astype(int)

plt.subplot(122)
plt.imshow(image2, cmap=plt.cm.gray)
```

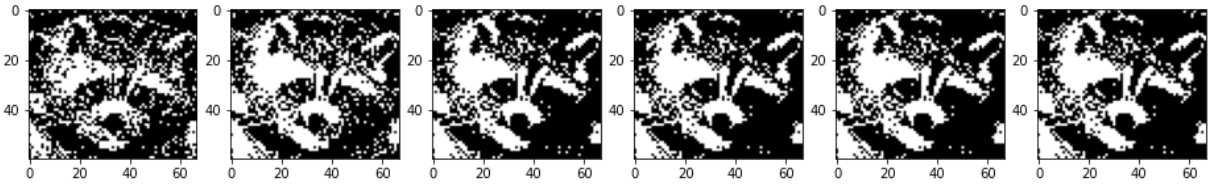
Out[ ]: <matplotlib.image.AxesImage at 0x7f33416ed0a0>



In [ ]:
```
# parameters
N = 2 # number of patterns
M = pattern.shape[0] # number of neurons
sparseness = np.sum(pattern)/len(pattern)
bias = 0.3 # theta
T = 5 # number of time steps

# plotting
fig, ax = plt.subplots(T//6+1,6, figsize=(16,(T//6+1)*3))
ax = ax.flatten()


# part 1
w = np.random.randn(M,M)*2 # random initialisation
w += np.outer(pattern-sparseness, pattern-sparseness)
pattern2 = image[:,::-1].flatten().astype(int)
w += np.outer(pattern2-sparseness, pattern2-sparseness)
w = w/N

# part 2
np.random.seed(42)
s = np.copy(pattern)
corrupt = np.random.rand(M)>0.7 # weight
s[corrupt] = pattern2[corrupt]
print(np.sum((s-pattern)**2)/len(pattern))
ax[0].imshow(s.reshape(image.shape),cmap=plt.cm.gray)

# part 3
for t in range(T):
    s = (0.5 + 0.5 * np.sign((w @ s) - bias)).astype(int)
```

```
    ax[t+1].imshow(s.reshape(image.shape),cmap=plt.cm.gray)
    print(np.sum((s-pattern)**2)/len(pattern))
```

```
0.12761194029850748
0.059203980099502486
0.003482587064676617
0.0
0.0
0.0
```



Design

Lets outline the method of what this program computes and what type of memory is simulated here.

This program computes the process of corrupting an image of a raccoon and testing the neuron M's ability to un-corrupt the image of the racoon using the Hopfield model of memory. It does so by first inserting the image and setting the plot parameters then (in Part 1) the code establishes a random initialization the weight matix. This random initization uses the np.outer() function to take the two outer products of the two vectors sparcsness and the pattern. This part of the code also establishes what pattern2 is, as a reflective image of the racoon. And then (in part 2) the code corrupts the random initialization of the file by a reshaping of the image based on the random amount of original M neural memory of the original raccoon weighted, which creates the value of sparcity of each of the patterns. And then(in part 3) fixes the corrupted image through the for loop. Modelling the gradual attempt at returning the image by summing the difference between the distributive multiplication of the image's current sparcity minus the bias theta(i), which outputs a new sparcity until the model has no more difference between its memory model and the corrupted image. The stored datasets of each image and its reshaped sparcity are fed to the model to simulate associative memory processes, as the ability to take an image that's slightly the same but different-- a reverse raccoon as used here --corrupting the original raccoon image. Such a model resembles how priming effects can plausibly identify a similar memory image and is also able to identify the relationship between the two and establish associative similarity that ends up being stored in the brain.

Analysis

Now lets look at the output this code generates (text and graphs).

The output of the code is the value of the np.sum((s-pattern)** 2)/len(pattern)), each number value as a representation of the sparcity of the test image in ascending order of all six graphs. Over the 5T - time steps of the for loop, each time calculates the sparcity of the patterns by the fraction of the elements==1, with the sparcity itself analysed by respectively multiplying the weight and sparceness minus the bias theta(i), and then returning the sum of the reshaped image according to s(t+1). This output is the sum of the image along every time step until the image has reverted to the original 0.0 and will remain there as it has no more reshaped corrupted pixel-array values to revert to the existing memory image.

Alterations

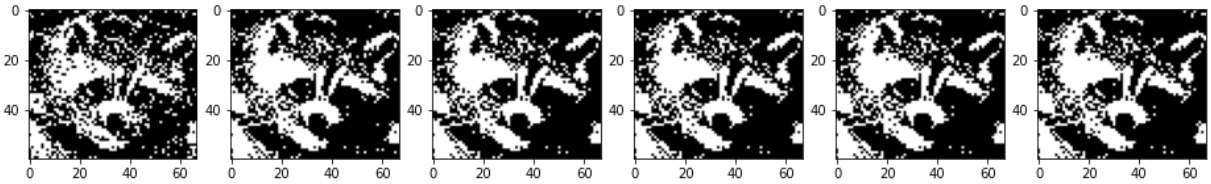
Lets expand on what happens when we change the relative weight of the test pattern.

When changing the test patterns relative weight to 1, the code outputs the original slay raccoon in all its uncorrupted beauty and glory :) . As the relative weight of the test pattern becomes very close to 0, the first image outputs the reflective raccoon as the majority image, and as the relative weight of the test pattern becomes very close to 1, the more of the original raccoon outputted in the first image. The weight of corruption is the difference amount of the original image valued out of 1.0 minus the weight of the test pattern, the remainder is the amount of the reflective raccoon corrupting the first image. This model has a memory, one of the original raccoon at its complete integral graph where the test pattern is 1.0 and, eventually uncorrupts the image through the loop back to its original.

In [ ]:
```
fig, ax = plt.subplots(T//6+1,6, figsize=(16,(T//6+1)*3))
ax = ax.flatten()
# part 1
w = np.random.randn(M,M)*2 # random initialisation
w += np.outer(pattern-sparseness, pattern-sparseness)
pattern2 = image[:,::-1].flatten().astype(int)
w += np.outer(pattern2-sparseness, pattern2-sparseness)
w = w/N
# part 2
np.random.seed(42)
s = np.copy(pattern)
corrupt = np.random.rand(M)>0.8 # weight
s[corrupt] = pattern2[corrupt]
print(np.sum((s-pattern)**2)/len(pattern))
ax[0].imshow(s.reshape(image.shape),cmap=plt.cm.gray)
# part 3
for t in range(T):
```

```
    s = (0.5 + 0.5 * np.sign((w @ s) - bias)).astype(int)
    ax[t+1].imshow(s.reshape(image.shape),cmap=plt.cm.gray)
    print(np.sum((s-pattern)**2)/len(pattern))
```

```
0.0845771144278607
0.009203980099502488
0.0002487562189054726
0.0
0.0
0.0
```



Discussion

There is good evidence that this model recapitulates at least some aspects of how memories are stotred in the brain. Lets examine which aspects of this model are biologically plausible, and which are not.

Sparcity is plausible because not every binary neuron is going to be firing at the same time. The analyzation of the images in terms of their array sparcity as the average of all the patterns, $s=<pn>N$.

Also this model recaptures the fundamentals of associative memory based on the ability to find its way back to the original memory image from partial cues that have been corrupted and are incomplete pieces of a memory image. This is a biologically plausible model because our minds often operate through priming effects to trigger our associative memories, we are able to identify information that resembles what we know/have seen before, even when this information is distorted or incomplete, we are still able to draw those associations back to the memory we already have.

This model does however, step by step fill in the integer differences along the pixelated graph which is something our minds usually vaguely fill in the blank with context we already may have. For example, if you see the side profile of a friends face, there is a good chance you are going to immediately identify your friend, but we don't need to automatically reconstruct what part of the face is incomplete in our mind in order to recognize them. Instead, we usually identify pieces that are the same (which this Hopfield model does) and then make assumptions based on the context of information available to identify someone. This model takes something it does not recognize and reconstructs the aspects of it to determine if the new image resembles the memory image, which in step-form is biologically implausible especially considering the

importance of friend vs. foe identification efficiency and storage of the memory image of the foe/friend.