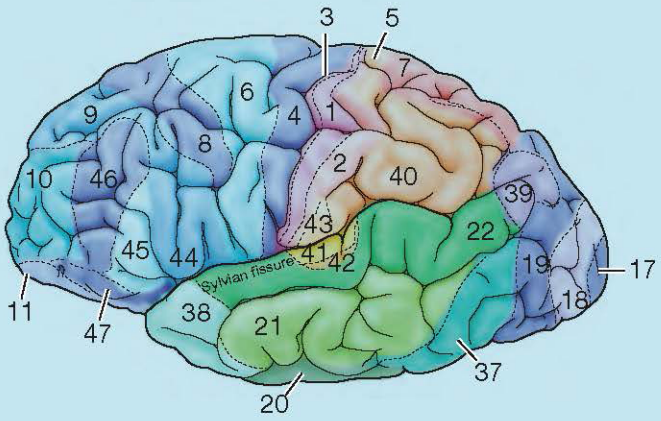
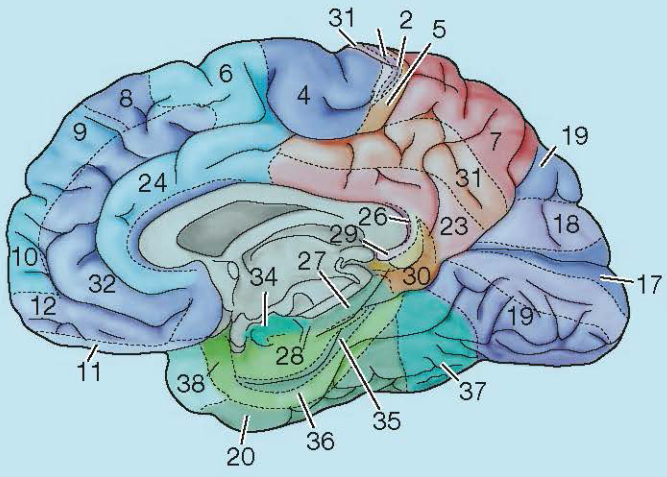
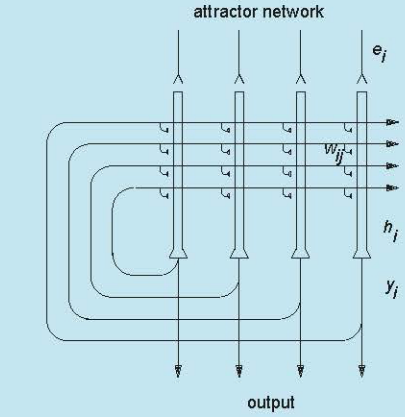
Brain Computations

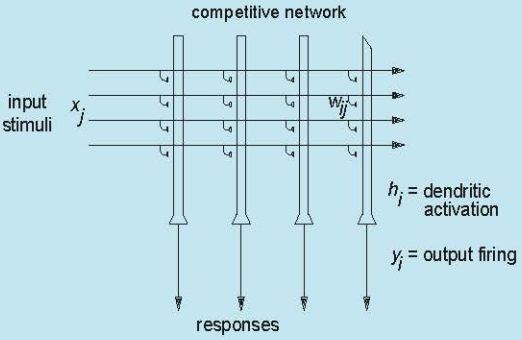
What and How

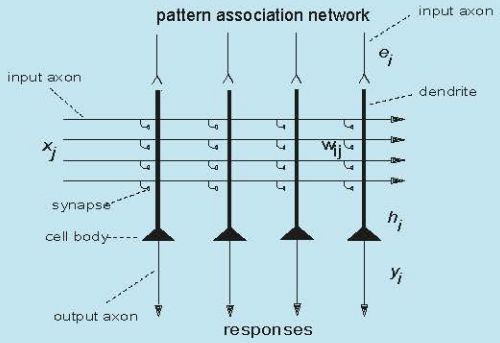
Edmund T. Rolls











OXFORD

Appendix 4 Simulation software for neuronal networks, and information analysis of neuronal encoding

D.1 Introduction

This Appendix of *Brain Computations: What and How* (Oxford University Press) (Rolls, 2021a) describes the Matlab software that has been made available with this book to provide simple demonstrations of the operation of some key neuronal networks related to cortical function. Previous versions of some of this software are provided in connection with *Cerebral Cortex: Principles of Operation* (Oxford University Press) (Rolls, 2016b) and *Emotion and Decision-Making Explained* (Oxford University Press) (Rolls, 2014a). The aim of providing the software is to enable those who are learning about these networks to understand how they operate and their properties by running simulations and by providing the programs. The code has been kept simple, but can easily be edited to explore the properties of these networks. The code itself contains comments that help to explain how the code works.

The programs are intended to be used with the descriptions of the networks and their properties provided in Appendix B of this book *Brain Computations: What and How* (Rolls, 2021a). Most of the papers cited in this Appendix can be found at https://www.oxcns.org. Exercises are suggested below that will provide insight into the properties of these networks, and will make the software suitable for class use. The code is available at https://www.oxcns.org. The programs are written in Matlab^{TM} (Mathworks Inc). The programs should work in any version of Matlab, whether it is running under Windows, Linux, or Apple software. The programs will also work under GNU Octave, which is available for free download. In addition, for those who use Python, Python versions of these programs are also provided at https://www.oxcns.org.

To get started, copy the Matlab program files PatternAssociationDemo.m, AutoAssociationDemo.m, CompetitiveNetDemo.m and SOMdemo.m, and the function files NormVeclen.m and spars.m, into a directory, and in Matlab cd into that directory.

The corresponding programs for Python are PatternAssociationDemo.py, AutoAssociationDemo.py, CompetitiveNetDemo.py and SOMdemo.py, and the function files NormVeclen.py and spars.py.

Sections D.2–D.4 describe these tutorial neuronal network programs.

In addition, Section D.6 describes tutorial software written in Matlab that illustrates some of the principles of operation of VisNet, the model of invariant visual object recognition described in Section 2.7 of this book (Rolls, 2021a) and by Rolls (2012d). The software is also available at https://www.oxcns.org.

Section D.7 describes code translated into Matlab that was used for the single neuron information analyses described by Rolls, Treves, Tovee and Panzeri (1997d) and the multiple single neuron information analyses described by Rolls, Treves and Tovee (1997b), and in Appendix C. The software is also available at https://www.oxcns.org.

Section D.8 describes Matlab code to illustrate the use of spatial view cells and allocentric

bearing to a landmark cells in Navigation described in Section 10.4.4 (Rolls, 2020b). The software is available at https://www.oxcns.org/software.

D.2 Autoassociation or attractor networks

The operation and properties of autoassociation (or attractor) neuronal networks are described in this book Rolls (2021a) Appendix B.3.

D.2.1 Running the simulation

In the command window of Matlab, after you have performed a 'cd' into the directory where the source files are, type 'AutoAssociationDemo' (with every command followed by 'Enter' or 'Return'). The program will run for a bit, and then pause so that you can inspect what has been produced so far. A paused state is indicated by the words 'Paused: Press any key' in the bottom of the Matlab window. Press 'Enter' to move to the next stage, until the program finishes and the command prompt reappears. You can edit the code to comment out some of the 'pause' statements if you do not want them, or to add a 'pause' statement if you would like the program to stop at a particular place so that you can inspect the results. To stop the program and exit from it, use 'Ctrl-C'. When at the command prompt, you can access variables by typing the variable name followed by 'Enter'. (Note: if you set '*display = 0*' near the beginning of the program, there will be no pauses, and you will be able to collect data quickly.)

The fully connected autoassociation network has N=100 neurons with the dendrites that receive the synapses shown as vertical columns in the generated figures, and $nSyn = N$ synapses onto each neuron. At the first pause, you will see a figure showing the 10 random binary training patterns with firing rates of 1 or 0, and with a *Sparseness* (referred to as a in the equations below) of 0.5. As these are binary patterns, the *Sparseness* parameter value is the same as the proportion of high firing rates or 1s. At the second pause you will see distorted versions of these patterns to be used as recall cues later. The number of bits that have been flipped $nFlipBits$=14.

Next you will see a figure showing the synaptic matrix being trained with the 10 patterns. Uncomment the pause in the training loop if you wish to see each pattern being presented sequentially. The synaptic weight matrix *SynMat* (the elements of which are referred to as w_{ij} in the equations below) is initialized to zero, and then each pattern is presented as an external input to set the firing rates of the neurons (the postsynaptic term), which because of the recurrent collaterals become also the presynaptic input. Each time a training pattern is presented, the weight change is calculated by a covariance learning rule

$$\delta w_{ij} = \alpha(y_i - a)(y_j - a) \tag{D.1}$$

where α is a learning rate constant, and y_j is the presynaptic firing rate. This learning rule includes (in proportion to y_i the firing rate of the ith postsynaptic neuron) increasing the synaptic weight if $(y_j - a) > 0$ (long-term potentiation), and decreasing the synaptic weight if $(y_j - a) < 0$ (heterosynaptic long-term depression). As these are random binary patterns with sparseness a (the parameter *Sparseness* in AutoassociationDemo.m), a is the average activity $< y_j >$ of an axon across patterns, and a is also the average activity $< y_i >$ of a neuron across patterns. In the exercises, you can try a form of this learning rule that is more biologically plausible, with only heterosynaptic long-term depression. The change of weight is added to the previous synaptic weight. In the figures generated by the code, the maximum weights are shown as white, and the minimum weights are black. Although both these rules

lead to some negative synaptic weights, which are not biologically plausible, this limitation can be overcome, as shown in the exercises. In the display of the synaptic weights, remember that each column represents the synaptic weights on a single neuron. The thin row below the synaptic matrix represents the firing rates of the neurons, and the thin column to the left of the synaptic weight matrix the firing rates of the presynaptic input, which are usually the same during training. Rates of 1 are shown as white, and of 0 as black.

Next, testing with the distorted patterns starts, with the output rates allowed to recirculate through the recurrent collateral axons for 9 recall epochs to produce presynaptic inputs that act through the synaptic weight matrix to produce the firing rate for the next recall epoch. The distorted recall cue is presented only on epoch 1 in what is described as the clamped condition, and after that is removed so that the network has a chance to settle into a perfect recall state representing the training pattern without being affected by the distorted recall cue. (In the cortex, this may be facilitated by the greater adaptation of the thalamic inputs than of the recurrent collaterals (Rolls, 2016b)). (Thus on epoch 1 the distorted recall cue is shown as the PreSynaptic Input column on the left of the display, and the Activation row and the Firing Rate row below the synaptic weight matrix are the Activations and Rates produced by the recall cue. On later epochs, the Presynaptic Input column shows the output firing Rate from the preceding epoch, recirculated by the recurrent collateral connections.)

We are interested in how perfect the recall is, that is how correlated the recalled firing rate state is with the original training pattern, with this correlation having a maximum value of 1. Every time you press 'Enter' a new recall epoch is performed, and you will see that over one to several recall epochs the recall usually will become perfect, that is the correlation, which is provided for you in the command window, will become 1. However, on some trials, the network does not converge to perfect recall, and this occurs if by chance the distorted recall cue happened to be much closer to one of the other stored patterns. Do note that with randomly chosen training patterns in relatively small networks this will sometimes occur, and that this 'statistical fluctuation' in how close some of the training patterns are to each other, and how close the distorted test patterns are to individual training patterns, is to be expected. These effects smooth out as the network become larger. (Remember that cortical neurons have in the order of 10,000 recurrent collateral inputs to each neuron, so these statistical fluctuations will be largely smoothed out. Do note also that the performance of the network will be different each time it is run, because the random number generator is set with a different seed on each run.) (Near the beginning of the program, you could uncomment the command that causes the same seed to be used for the random number generator for each run, to help with further program development that you may try.)

After the last test pattern has been presented, the percentage of the patterns that were correctly recalled is shown, using as a criterion that the correlation of the recalled firing rate with the training pattern $r >= 0.98$. For the reasons just described related to the random generation of the training and distorted test patterns, the percentage correct will vary from run to run, so taking the average of several runs is recommended.

Note that the continuing stable firing after the first few recall epochs models the implementation of short-term memory in the cerebral cortex (Rolls, 2008d, 2016b, 2021a).

Note that the retrieval of the correct pattern from a distorted version that includes missing 1s models completion in the cerebral cortex, which is important in episodic memory as implemented in the hippocampus in which the whole of the episodic memory can be retrieved from any part (Rolls, 2008d, 2016b; Kesner and Rolls, 2015) (Chapter 9).

Note that the retrieval of the correct pattern from a distorted version that includes missing 1s or has 1s instead of 0s models correct memory retrieval in the cerebral cortex and hippocampus even if there is some distortion of the recall cue (Rolls, 2016b, 2021a). This also enables generalization to similar patterns or stimuli that have been encountered previously,

which is highly behaviourally adaptive (Rolls, 2016b, 2021a). These processes are also important in completion in episodic memory as implemented in the hippocampus in which the whole of the episodic memory can be retrieved from any part (Rolls, 2008d, 2016b; Kesner and Rolls, 2015) (Chapter 9).

D.2.2 Exercises

1. Measure the percent correct as you increase the number of patterns from 10 to 15 with the Sparseness remaining at 0.5. Plot the result. How close is your result to that found by Hopfield (1982), which was $0.14N$ for a sparseness a of 0.5? You could try increasing N by 10 (to 1000) and multiplying the number of training patterns by 10 and increasing $nFlipBits$ to say 100 to see whether your results, with smaller statistical fluctuations, approximate more closely to the theoretical value.

2. Test the effect of reducing the Sparseness (a) to 0.1, which will increase the number of patterns that can be stored and correctly recalled, i.e. the capacity of the network (Treves and Rolls, 1991; Rolls, 2016b, 2021a). (Hint: try a number of patterns in the region of 30.) You could also plot the capacity of the network as a function of the Sparseness (a) with values down to say 0.01 in larger networks, with theoretical results provided by Treves and Rolls (1991) and Rolls (2021a).

3. Test the effect of altering the learning rule, from the covariance rule to the rule with heterosynaptic long term depression (LTD), which is as follows (Rolls, 2021a):

$$\delta w_{ij} = \alpha(y_i)(y_j - a). \tag{D.2}$$

(Hint: in the training loop in the program, at about line 102, comment in the heterosynaptic LTD rule.)

4. Test whether the network operates with positive-only synaptic weights. (Hint: there are two lines (close to 131) just after the training loop that if uncommented will add a constant to all the numbers in the synaptic weight matrix to make all the numbers positive (with the minimum synaptic weight clipped at 0).

5. What happens to the recall if you train more patterns than the critical capacity? (Hint: see Rolls (2021a) section B.16.)

6. Does it in practice make much difference if you allow self-connections, especially in large networks? (Hint: in the program in the training loop if you comment out the line 'if syn $\sim=$ neuron' and its corresponding 'end', self-connections will be allowed.)

D.3 Pattern association networks

The operation and properties of pattern association networks are described in Rolls (2021a) Appendix B.2.

D.3.1 Running the simulation

In the command window of Matlab, after you have performed a 'cd' into the directory where the source files are, type 'PatternAssociationDemo'. The program will run for a bit, and then

pause so that you can inspect what has been produced so far. A paused state is indicated by the words 'Paused: Press any key'. Press 'Enter' to move to the next stage, until the program finishes and the command prompt reappears. You can edit the code to comment out some of the 'pause' statements if you do not want them, or to add a 'pause' statement if you would like the program to stop at a particular place so that you can inspect the results. To stop the program and exit from it, use 'Ctrl-C'. When at the command prompt, you can access variables by typing the variable name followed by 'Enter'. (Note: if you set 'display = 0' near the beginning of the program, there will be few pauses, and you will be able to collect data quickly.)

The fully connected pattern association network has N=8 output neurons with the dendrites that receive the synapses shown as vertical columns in the generated figures, and $nSyn$=64 synapses onto each neuron. At the first pause, you will see the 8 random binary conditioned stimulus (CS) training patterns with firing rates of 1 or 0, and with a sparseness of 0.25 (*InputSparseness*). As these are binary patterns, the sparseness is the same as the proportion of high firing rates or 1s. At the second pause you will see distorted versions of these patterns to be used as recall cues later. The number of bits that have been flipped $nFlipBits$=8.

Next you will see the synaptic matrix being trained with the 8 pattern associations between each conditioned stimulus (CS) and its unconditioned stimulus (US), the output firing. In this simulation, the default is for each US pattern to have one bit on, so that the sparseness of the US, *OutputSparseness* is $1/N$. Uncomment the pause in the training loop if you wish to see each pattern being presented sequentially. The synaptic weight matrix *SynMat* (elements of which are referred to as w_{ij} below) is initialized to zero, and then each pattern pair is presented as a CS and US pair, with the CS specifying the presynaptic input, and the US the firing rate in the N neurons present when the CS is presented. Each time a training pattern pair is presented, the weight change is calculated by an associative learning rule that includes heterosynaptic long term depression

$$\delta w_{ij} = \alpha(y_i)(x_j - a). \tag{D.3}$$

where α is a learning rate constant, and x_j is the presynaptic firing rate. This learning rule includes (in proportion to y_i the firing rate of the i^{th} postsynaptic neuron produced by the US) increasing the synaptic weight if $(y_j - a) > 0$ (long-term potentiation), and decreasing the synaptic weight if $(x_j - a) < 0$ (heterosynaptic long-term depression). As the CS patterns are random binary patterns with sparseness a (the parameter *InputSparseness*), a is the average activity $< x_j >$ of an axon across patterns. The reasons for the use of this learning rule are described in Rolls (2021a) Section B.3.3.6. In the exercises, you can try a Hebbian associative learning rule that does not include this heterosynaptic long-term depression. The change of weight is added to the previous synaptic weight. In the figures generated by the code, the maximum weights are shown as white, and the minimum weights are black. Although both these rules lead to some negative synaptic weights, which are not biologically plausible, this limitation can be overcome, as shown in the exercises. In the display of the synaptic weights, remember that each column represents the synaptic weights on a single neuron. The top thin row below the synaptic matrix represents the activations of the neurons, the bottom thin row the firing rates of the neurons, and the thin column to the left of the synaptic weight matrix the firing rates of the presynaptic input, i.e. the recall (CS) stimulus. Rates of 1 are shown as white, and of 0 as black.

Next, testing with the distorted CS patterns starts. We are interested in how the pattern association network generalizes with distorted inputs to produce the correct US output that should be produced if there were no distortion. The recall performance is measured by how correlated the recalled firing rate state (the Conditioned Response, CR) is with the output

that should be produced to the original training CS pattern, with this correlation having a maximum value of 1. Every time you press 'Enter' a new CS is presented. Do note that with randomly chosen distortions of the CS patterns to produce the recall cue, in relatively small networks the distorted CS will sometimes be close of another CS, and that this 'statistical fluctuation' in how close some of the distorted CSs are to the original CSs is to be expected. These effects smooth out as the network become larger. (Remember that cortical neurons have in the order of 10,000 inputs to each neuron (in addition to the recurrent collaterals, Rolls (2021a)), so these statistical fluctuations will be largely smoothed out. Do note also that the performance of the network will be different each time it is run, because the random number generator is set with a different seed on each run. (Near the beginning of the program, you could uncomment the command that causes the same seed to be used for the random number generator for each run, to help with further program development that you may try.)

After the last distorted testing CS pattern has been presented, the percentage of the patterns that were correctly recalled is shown, using a criterion for the correlation of the recalled firing rate with the training pattern $r >=$0.98. For the reasons just described related to the random generation of the distorted test patterns, the percentage correct will vary from run to run, so taking the average of several runs is recommended.

Note that the retrieval of the correct output (conditioned response, CR) pattern from a distorted CS recall stimulus models generalization in the cerebral cortex, which is highly behaviourally adaptive (Rolls, 2016b, 2021a).

D.3.2 Exercises

1. Test how well the network generalizes to distorted recall cues, by altering $nFlipBits$ to values between 0 and 14. Remember to take the average of several simulation runs.

2. Test the effect of altering the learning rule, from the rule with heterosynaptic long term depression (LTD) as in Eqn. D.3, to a simple associative synaptic modification rule which is as follows (Rolls, 2021a):

$$\delta w_{ij} = \alpha(y_i)(x_j). \tag{D.4}$$

(Hint: in the training loop in the program, at about line 110, comment out the heterosynaptic LTD rule, and uncomment the associative rule.)

If you see little difference, can you suggest conditions under which there may be a difference, such as higher loading? (Hint: check Rolls (2021a) section B.2.7.)

3. Describe how the threshold non-linearity in the activation function helps to remove interference from other training patterns, and from distortions in the recall cue. (Hint: compare the activations to the firing rates of the output neurons.)

4. In the pattern associator, the conditioned stimulus (CS) retrieves the unconditioned response (UR), that is the effects produced by the unconditioned stimulus (US) with which it has been paired during learning. Can the opposite occur, that is, can the US be used to retrieve the CS? If not, in what type of network is the retrieval symmetrical? (Hint: if you are not sure, read Rolls (2021a) Sections B.2 and B.3.)

5. The network simulated uses local representations for the US, with only one neuron on for each US. If you can program in Matlab, can you write code that would generate distributed representations for the US patterns? Then can you produce new CS patterns with random binary representations and sparsenesses in the range 0.1–0.5, like those used for the

autoassociator, so that you can then investigate the storage capacity of pattern association networks. (Hint: reference to Rolls (2021a) section B.2.7 and B.2.8 may be useful.)

D.4 Competitive networks and Self-Organizing Maps

The operation and properties of competitive networks are described in Rolls (2021a) Appendix B.4. This is a self-organizing network, with no teacher. The input patterns are presented in random sequence, and the network learns how to categorise the patterns, placing similar patterns into the same category, and different patterns into different categories. The particular neuron or neurons that represent each category depend on the initial random synaptic weight vectors as well as the learning, so the output neurons are different from simulation run to simulation run. Self-organizing maps are investigated in Exercise 4.

D.4.1 Running the simulation

In the command window of Matlab, after you have performed a 'cd' into the directory where the source files are, type 'CompetitiveNetDemo'. The program will run for a bit, and then pause so that you can inspect what has been produced so far. A paused state is indicated by the words 'Paused: Press any key'. Press 'Enter' to move to the next stage, until the program finishes and the command prompt reappears. You can edit the code to comment out some of the 'pause' statements if you do not want them, or to add a 'pause' statement if you would like the program to stop at a particular place so that you can inspect the results. To stop the program and exit from it, use 'Ctrl-C'. When at the command prompt, you can access variables by typing the variable name followed by 'Enter'. (Note: if you set '*display = 0*' near the beginning of the program, there will be few pauses, and you will be able to collect data quickly.)

The fully connected competitive network has N=100 output neurons with the dendrites that receive the synapses shown as vertical columns, and $nSyn$=100 synapses onto each neuron. With the display on (*display = 1*), at the first pause, you will see the 28 binary input patterns for training with firing rates of 1 or 0. Each pattern is 20 elements long, and each pattern is shifted down 3 elements with respect to the previous pattern, with the parameters supplied. (Each pattern thus overlaps in 17 / 20 locations with its predecessor.)

Next you will see the synaptic matrix being trained with the 28 input patterns, which are presented 20 times each (*nepochs* = 20) in random permuted sequence. Uncomment the pause in the training loop if you wish to see each pattern being presented sequentially. The synaptic weight matrix *SynMat* is initialized so that each neuron has random synaptic weights, with the synaptic weight vector on each neuron normalized to have a length of 1. Normalizing the length of the vector during training is important, for it ensures that no one neuron increases its synaptic strengths to very high values and thus wins for every input pattern. (Biological approximations to this are described by Rolls (2021a) in section B.4.9.2, and involve a type of heterosynaptic long-term depression of synaptic strength that is large when the postsynaptic term is high, and the presynaptic firing is low.) (In addition, for convenience, each input pattern is normalized to a length of 1, so that when the dot product is computed with a synaptic weight vector on a neuron, the maximum activation of a neuron will be 1.)

The firing rate of each neuron is computed using a binary threshold activation function (which can be changed in the code to threshold linear). The sparseness of the firing rate representation can be set in the code using *OutputSparseness*, and as supplied is set to 0.01 for the first demonstration, which results in one of the N=100 output neurons having a high firing

rate. Because the (initially random) synaptic weight vectors of each neuron are different, the activations of each of the neurons will be different.

Learning then occurs, using an associative learning rule with synapses increasing between presynaptic neurons with high rates and postsynaptic neurons with high rates. Think of this as moving the synaptic weight vector of a high firing neuron to point closer to the input pattern that is making the neuron fire. At the same time, the other synapses on the same neuron become weaker due to the synaptic weight normalization, so the synaptic weight vector of the neuron moves away from other, different, patterns. In this way, the neurons self-organize so that some neurons point towards some patterns, and other neurons towards other patterns. You can watch these synaptic weight changes during the learning in the figure generated by the code, in which some of the synapses will become stronger (white) on a neuron and correspond to one or several of the patterns, and at the same time the other synapses on the neuron will become weaker (black). In the display of the synaptic weights, remember that each column represents the synaptic weights on a single neuron. The top thin row below the synaptic matrix represents the activations of the neurons, the bottom thin row the firing rates of the neurons, and the thin column to the left of the synaptic weight matrix the firing rates of the presynaptic input, i.e. the pattern currently being presented. Rates of 1 are shown as white, and of 0 as black. Neurons that remain unallocated, and do not respond to any of the input patterns, appear as vertical columns of random synaptic weights onto a neuron. These neurons remain available to potentially categorise different input patterns.

Next, testing with the different patterns starts. We are interested in whether similar input patterns activate the same output neurons; and different input patterns different output neurons. This is described as categorisation (Rolls, 2021a). With the display on (*display* = 1), each time you press 'Enter' the program will step through the patterns in sequence. You will notice in the synaptic matrix that with these initial parameters, only a few neurons have learned. When you step through each testing pattern in sequence, you will observe that one of the neurons responds to several patterns in the sequence, and these patterns are quite closely correlated with each other. Then another neuron becomes active for a further set of close patterns. This is repeated as other patterns are presented, and shows you that similar patterns tend to activate the same output neuron or neurons, and different input patterns tend to activate different output neurons. This illustrates by simulation the categorisation performed by competitive networks.

To illustrate these effects more quantitatively, after the patterns have been presented, you are shown first the correlations between the input patterns, and after that the correlations between the output firing rates produced by each pattern. These results are shown as correlation matrices. You should observe that with the network parameters supplied, the firing rates are grouped into several categories, each corresponding to several input patterns that are highly correlated in the output firing that they produce. The correlation matrix between the output firing rate for the different patterns thus illustrates how a competitive network can categorise similar patterns as similar to each other, and different patterns as different.

D.4.2 Exercises

1. Test the operation of the system with no learning, most easily achieved by setting *nepochs* = 0. This results in the input patterns being mapped to output rates through the random synaptic weight vectors for each neuron. What is your interpretation of what is shown by the correlation matrix between the output firing rates?

(Hint: the randomizing effect of the random synaptic weight vectors on each neuron is to separate out the patterns, with many of the patterns having no correlation with other patterns. In this mode, pattern separation is produced, and not categorisation in that similar inputs

are less likely to produce similar outputs, a key property in perception and memory (Rolls, 2021a). Pattern separation is important in its own right as a process (Rolls, 2016e), and is implemented for example by the random non-associatively modifiable connections of the dentate mossy fibre synapses onto the CA3 neurons (Treves and Rolls, 1992; Rolls, 2016b, 2013b; Kesner and Rolls, 2015; Rolls, 2021a).)

2. Investigate and describe the effects of altering the similarity between the patterns on the categorization, by for example altering the value of *shift* in the range 2–20.

3. Investigate and describe the effects of making the output firing rate representation (*OutputSparseness*) less sparse. It is suggested that values in the range 0.01–0.1 are investigated.

4. If there is lateral facilitation of nearby neurons, which might be produced by short-range recurrent collaterals in the cortex, then self-organizing maps can be generated, providing a model of topographic maps in the cerebral cortex (Section B.4.6 and Rolls (2021a) Section B.4.6). Investigate this with the modified competitive network 'SOMdemo'. The map is made clear during the testing, in which the patterns are presented in order, and the neurons activated appear in different mapped positions in the firing rate array. Run the program several times to show that the details of the map are different each time, but that the principles are the same, that nearby neurons tend to respond to similar patterns, and that singularities (discontinuities) in the map can occur, just as are found in the primary visual cortex, V1.

The important change of 'SOMdemo' from 'CompetitiveNetDemo' is a local spatial filter for the firing rate array to simulate the effect of facilitation of the firing rates of nearby neurons by the short-range excitatory recurrent collateral connections between nearby cortical pyramidal cells. This is implemented by the *SpatialFilter* kernel the range of which is set by *FilterRange*. Investigate the effects of altering *FilterRange* from its default value of 11 on the topographic maps that are formed. Other changes to produce 'SOMdemo' include modifying the *OutPutSparseness* of the Firing Rate representation to 0.3. Investigate the effect of modifying this on the maps being formed. Another alteration was to alter the activation function from binary threshold to linear threshold to enable graded firing rate representations to be formed.

What are the advantages of the representations formed when self-organizing maps are present? (Hint: Think of interpolation of untrained input patterns close to those already trained; and check Rolls (2021a) Section B.4.6.)

D.5 Further developments

The neuronal network simulation code supplied and described above may provide a route for readers to develop their own programs. Further examples of Matlab code used to investigate neural systems are available in Anastasio (2010). Matlab itself has a Neural Network Toolbox, with an introduction to its use in modelling simple networks provided in Wallisch et al. (2009).

D.6 Matlab code for a tutorial version of VisNet

Tutorial software written in Matlab that illustrates some of the principles of operation of VisNet, the model of invariant visual object recognition described in Section 2.7 of this book (Rolls, 2021a) and by Rolls (2012d), is available at https://www.oxcns.org/software. This is

not the full version of VisNet used for most of the research described in Section 2.7, but it may be useful to illustrate some of the principles of operation of VisNet.

D.7 Matlab code for information analysis of neuronal encoding

Code translated into Matlab that was used for the single neuron information analyses described by Rolls, Treves, Tovee and Panzeri (1997d) and the multiple single neuron information analyses described by Rolls, Treves and Tovee (1997b), and in Appendix C, is also available at https://www.oxcns.org/software.

D.8 Matlab code to illustrate the use of spatial view cells in navigation

Matlab code to illustrate the use of spatial view cells (NavSVC.m), allocentric bearing to a landmark cells (NavABL.m), and combinations of allocentric bearing to a landmark cells using triangulation for navigation (NavTRI.m) described by (Rolls, 2020b) and in Section 10.4.4 is also available at https://www.oxcns.org/software.

D.9 Highlights

1. Appendix D describes Matlab software that has been made available with *Brain Computations: What and How* (Rolls, 2021a) at https://www.oxcns.org to provide simple demonstrations of the operation of some key neuronal networks related to cortical function.
2. The software demonstrates the operation of pattern association networks, autoassociation / attractor networks, competitive networks, and self-organizing maps.
3. The availability of software to illustrate the operation of VisNet, a model of invariant visual object recognition, has also been described.
4. The availability of software to analyse the encoding of information by single neurons, and by populations of single neurons, has also been described.