Computational Neuroscience 2

USN: 303039534

1 Unsupervised learning

For a general many-input-one-output network we can define an input vector $\mathbf{u}$ as the activity of the neurons in the input layer and a weight vector $\mathbf{w}$ as the strength of connections between the input neurons and the output neuron. The total input to the output neuron is thus $v = \mathbf{w} \cdot \mathbf{u}$.

Based on Hebb's conjecture from 1949, we can write a simple learning rule where the strength of a synapse increases with correlated pre- and post-synaptic coactivity:

$$\tau_w \frac{d\mathbf{w}}{dt} = v\mathbf{u} \tag{1}$$

If we assume that synaptic weight changes occur over a slower timescale than the inputs, we can use 'batch learning' and average over all input data in every learning step: $\tau_w \frac{d\mathbf{w}}{dt} = \langle v\mathbf{u} \rangle$. Since $v = \mathbf{w} \cdot \mathbf{u}$ and defining the correlation matrix $\mathbf{Q} = \langle \mathbf{uu} \rangle$, this allows us to learn weights using simple matrix multiplication

$$\tau_w \frac{d\mathbf{w}}{dt} = \mathbf{Q} \cdot \mathbf{w} \tag{2}$$

Note that this definition of 'correlation' is slightly odd, as for example the self correlation is given by $Q_{ii} = \langle u_i^2 \rangle$ rather than self correlations always being 1. We then continue to iterate over equation 2 until the weights have converged. If the activities of our neurons correspond to firing rates, all u_i must be positive. The correlation matrix is therefore a non-negative matrix and we cannot learn negative correlations. Instead, it is therefore common to use a covariance-based learning rule where we let $\mathbf{u}' = \mathbf{u} - \langle \mathbf{u} \rangle$. Now defining the covariance matrix $\mathbf{C} = \langle (\mathbf{u} - \langle \mathbf{u} \rangle)(\mathbf{u} - \langle \mathbf{u} \rangle) \rangle$ we can rewrite our learning rule as

$$\tau_w \frac{d\mathbf{w}}{dt} = \mathbf{C} \cdot \mathbf{w} \tag{3}$$

We note that since these batch learning rules are linear in $\mathbf{w}$, we can treat them analytically. At any given time, we can decompose the weight vector in terms of the eigenvectors $\mathbf{e}_\mu$ of $\mathbf{C}$:

$$\mathbf{w}(t) = \sum_\mu c_\mu(t)\mathbf{e}_\mu \tag{4}$$

Solving the linear differential equation 3 we get $c_\mu = \exp\left(\frac{\lambda_\mu t}{\tau_w}\right)(\mathbf{w}(0) \cdot \mathbf{e}_\mu)$, and putting these results together gives

$$\mathbf{w}(t) = \sum_\mu \exp\left(\frac{\lambda_\mu t}{\tau_w}\right)(\mathbf{w}(0) \cdot \mathbf{e}_\mu)\mathbf{e}_\mu \tag{5}$$

If the covariance matrix is non-degenerate, we expect the eigenvector corresponding to the highest eigenvalue to dominate this sum at long times due to the coefficient being exponential in the eigenvalue. This in turn suggests that the weight vector will converge towards this first principal component unless this is orthogonal to the initial weight vector. However, this basic Hebbian learning will lead to unphysiological unbounded weight increases and is therefore usually augmented with some form of normalization which can alter the simple behavior expected from equation 5 as will be evident from the next two sections.

We note that since the timescale is arbitrary for the above equations, we can set $\tau_w = 1$ for the remainder of this section without loss of generality. We require dt to be smaller than τ_w for accurate Euler integration and therefore let dt=0.01. This is equivalent to using a discrete learning rule with $\epsilon = \frac{dt}{\tau_w} = 0.01$.

1.1 Multiplicative normalization

Multiplicative normalization is commonly used, where we subtract a term from the simple Hebbian learning rule proportional to the current weights at each iteration. We implement this using Oja's learning rule:

$$\tau_w \frac{d\mathbf{w}}{dt} = v\mathbf{u} - \alpha v^2 \mathbf{w} \tag{6}$$

To see how this leads to normalization, we take the dot product of equation 6 with $\mathbf{w}$

$$\tau_w \frac{d|\mathbf{w}|^2}{dt} = v\mathbf{w}\cdot\mathbf{u} - \alpha v^2 \mathbf{w}\cdot\mathbf{w} = v^2(1-\alpha|\mathbf{w}|^2) \tag{7}$$

Hence $\frac{d|\mathbf{w}|^2}{dt} \to 0$ as $\alpha|\mathbf{w}|^2 \to 1$ and Oja normalization constrains the modulus of the weight vector. At long times, we thus converge towards a weight vector of magnitude $|\mathbf{w}| \to \frac{1}{\sqrt{\alpha}}$. In the following, we let $\alpha = 1$ such that the weight vector converged to is normalized.

As above, we average over the training inputs and get a batch learning rule

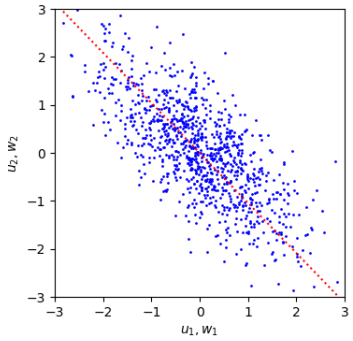
$$\tau_w \frac{d\mathbf{w}}{dt} = \mathbf{Q}\cdot\mathbf{w} - \alpha(\mathbf{w}\cdot\mathbf{Q}\cdot\mathbf{w})\mathbf{w} \tag{8}$$

We then update weights according to equation 8 until the norm of the change in weights between iterations is $\epsilon < 10^{-6}$. Setting $\alpha = 1$ we note that when $\mathbf{w}$ becomes an eigenvector of $\mathbf{Q}$ with eigenvalue λ, we can write $\mathbf{Q}\cdot\mathbf{w} = \lambda\mathbf{w}$ and $(\mathbf{w}\cdot\mathbf{Q}\cdot\mathbf{w}) = \lambda$, giving $\tau_w \frac{d\mathbf{w}}{dt} = 0$. For Oja's rule, we thus expect the weight vector to converge to an eigenvector of $\mathbf{Q}$, and we expect this to be the largest eigenvector on the basis of equation 5.

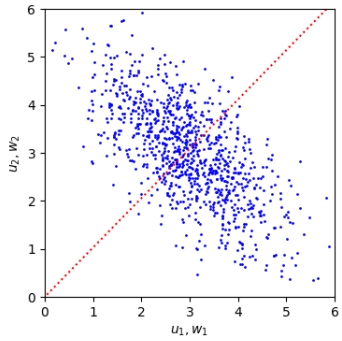
We see an example of this in figure 1a where we train the network on a two-dimensional Gaussian dataset with each point corresponding to a two-dimensional input datapoint $\mathbf{u}$. The red line indicates the direction of the final weight vector, shifted vertically to run through the mean of the dataset, and we see that it does indeed align with the first principal component (the axis of maximum variability).

However, if we shift the mean of the Gaussian input dataset to (3,3), the correlation matrix $\mathbf{Q}$ becomes a positive matrix and the largest eigenvalue now corresponds to the eigenvector (1,1) although there is still a negative covariance between the x and y component of the data (figure 1b). In this case, the converged weight vector thus ends up being perpendicular to the first principal component of the data.

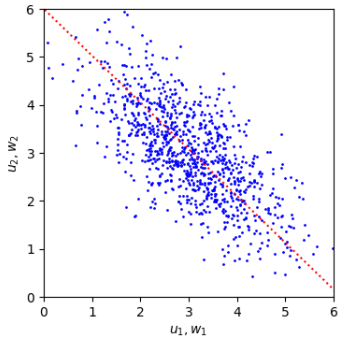
However, we can salvage this behavior by using the covariance-based learning rule from equation 3 on the same dataset (figure 1c). This allows us to once again capture the negative covariance between the components u_1 & u_2 as the converged weight vector aligns with the first principal component of the data.



(a) Mean (0,0) with correlation-based training. Final weight vector: [0.692, -0.722]. Principal components: [0.692, -0.722] & [0.722, 0.692].



(b) Mean (3,3) with correlation-based training. Final weight vector: [0.697, 0.717]. Principal components: [0.696, -0.718] & [0.718, 0.696].



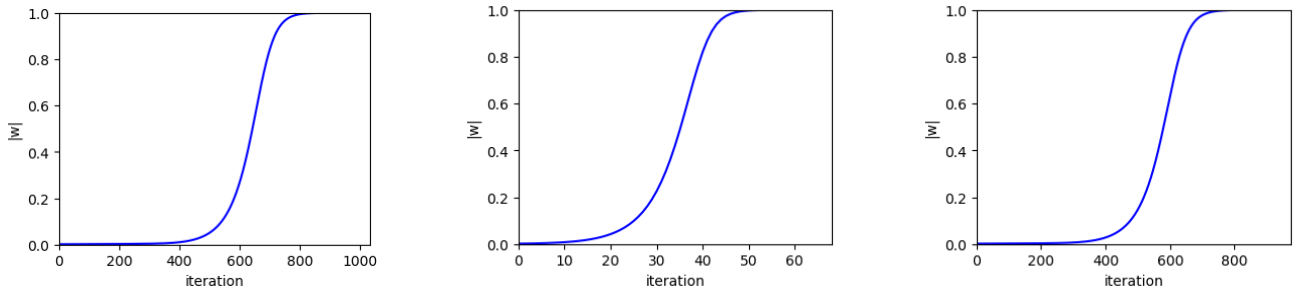
(c) Mean (3,3) with covariance-based training. Final weight vector: [0.716, -0.698]. Principal components: [0.716, -0.698] & [0.698, 0.716].

Figure 1: Learning weight vectors from 500 input data points with Oja's rule. In each case, we start from [w1, w2] = [0.001, 0.001] and iterate through equation 8 until convergence using either a correlation or covariance-based learning rule. Red dotted lines indicate the direction of the final weight vector. All datasets were generated with a slope of 1, a correlation of $\rho = -0.7$, and standard deviations of 1 and $1-\rho^2$ in the u_1 and u_2 directions.

If instead we were to have a positive covariance between the u_1 and u_2 components of our input vectors, the final weight vector aligns with the principal component irrespective of whether the mean of our distribution is zero, positive or negative provided that the mean of both the u_1 and u_2 components have the same sign. In this case we instead fail to recover the first principal component in the case where the mean of u_1 is positive while the mean of $u2$ is negative

or vice versa. In that case we get a final vector of [0.692, -0.722] despite a first principal component of [0.689, 0.725] in an example simulation. This is again salvaged by using a covariance-based learning rule.

We can also investigate the magnitude of our weight vectors over time since we expect these to converge smoothly to 1 as discussed above, and we see that this is indeed the case (figure 2). Interestingly, it appears that convergence to the [1,1] vector in our second simulation is much faster than converging to a [1, -1] vector in simulations 1 & 3.



(a) (b) (c)

Figure 2: Weight vector magnitude as a function of iteration number for the three simulations in figure 1.

1.2 Subtractive normalization

A different form of normalization that is commonly used to prevent excessive growth of weights is subtractive normalization as specified in equation 9.

$$\tau_w \frac{d\mathbf{w}}{dt} = v\mathbf{u} - \frac{v(\mathbf{n}\cdot\mathbf{u})\mathbf{n}}{N_u} \tag{9}$$

This constrains the sum of the weights to be constant as can be proven by taking the dot product with $\mathbf{n}$ where $\mathbf{n}$ is a vector of ones:

$$\tau_w \frac{d\mathbf{n}\cdot\mathbf{w}}{dt} = \tau_w \frac{d\sum_i w_i}{dt} = v\mathbf{n}\cdot\mathbf{u}(1 - \frac{\mathbf{n}\cdot\mathbf{n}}{N_u}) = 0 \tag{10}$$

The total sum of the weights in the system is thus constant, but these weights must still be thresholded to avoid unbounded growth of complementary weights towards $\pm\infty$. This is commonly achieved by constraining all weights to be ≥ 0 in which case there is an upper saturation limit of $w_{max} = \sum_i w_i(0)$.

We once again average our learning rule from equation 9 over the training data and get

$$\tau_w \frac{d\mathbf{w}}{dt} = \mathbf{Q}\cdot\mathbf{w} - \frac{(\mathbf{w}\cdot\mathbf{Q}\cdot\mathbf{n})\mathbf{n}}{N_u} \tag{11}$$

This is a highly competitive learning rule, and we therefore observe convergence to weights of [1,0] or [0,1] depending on initial conditions in the vast majority of cases, including the three datasets in figure 1 and equivalent datasets with positive ρ (data not shown). Even using identical initial weights, we will still converge to [0,1] or [1,0] rather than obeying the symmetry of the initial weights, since the data is noisy which leads to our covariance and correlation matrices having non-identical diagonal elements and therefore to symmetry breaking.

However, there exist a few special cases where we observe different behavior or where the weights still align with the principal components. One such case is where the data has a principal component that is either [0,1] or [1,0]. As an example, we alter our 2d Gaussian function to generate data with a slope of 0 and rho=0.995. The correlation matrix now has principal components of [1.0, 0.0] & [0.0, 1.0] and our weights evolve from [0.5, 0.5] to [1.0, 0] over the course of the simulation. In this case we thus do recover our first principal component.

Another special case is when the system is initialized at a fixed point. The fixed point of this system is given by

$$w_2 = \frac{Q_{11} - Q_{12}}{Q_{22} - Q_{12}} w_1 \tag{12}$$

This is an unstable fixed point, but if we initialize the weights at exactly this ratio they will remain there indefinitely due to both weight derivatives being 0. If this vector matches the first principal component of the data, the long term

behavior of the weight vector will also be to match this principal component. This is achieved when u_1 and u_2 are positively correlated with a slope of 1 and equal autocorrelations such that $w_1 = w_2$ is both the fixed point and first principal component.

In these two cases, we thus observe produce a weight vector aligned with the principal component axis of the dataset, and otherwise we do not. To illustrate this, we run 500 simulations with random initial vectors constrained by $w_1 + w_2 = 1$ and $w_1, w_2 \in [0, 1]$, and with the data having $\rho = 0.7$ and a slope of either 1 or 0.2. We then compare the converged weight vector to the first principal component of the input data by calculating the angle θ between the two vectors. We plot histograms of these angles in figure 3.

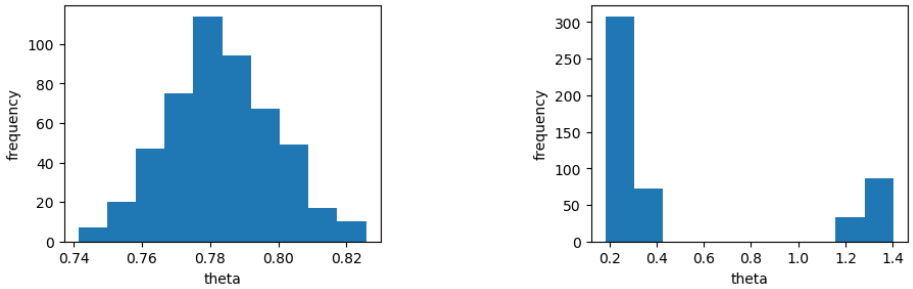


Figure 3: Histograms of angles between the first PC vector and the weight vector converged to from a random initial vector when the data has a slope of 1 (a) or 0.2 (b). We do not recover the first principal component since $\theta \neq 0$.

We see that there is very little variability in the long-term behavior of the system. In the case of a slope of 1 when the axis of the principal component is on average exactly between the [0,1] and [1,0] vectors, we arrive at a narrow distribution of angles near $\theta = \pi/4$. When the slope is 0.2, we get two narrow distributions near $\theta = 0.20$ corresponding to a final weight vector of [1,0] and near $\theta = 1.37$ corresponding to a final weight vector of [0,1].

2 Ocular dominance columns

In the following, we consider a many-input-many-ouput recurrent circuit with activity vector $\mathbf{v}$, input vector $\mathbf{u}$, feedforward weight matrix $\mathbf{W}$ and thus total input vector $\tilde{\mathbf{u}} = \mathbf{W} \cdot \mathbf{u}$. We additionally let the system have a fixed recurrent weight matrix $\mathbf{M}$. The dynamics of this system follow differential equation 13

$$\tau_r \frac{d\mathbf{v}}{dt} = -\mathbf{v} + \mathbf{W} \cdot \mathbf{u} + \mathbf{M} \cdot \mathbf{v} \tag{13}$$

Provided that the eigenvalues of $\mathbf{M}$ have real parts < 1, a stability analysis shows that this system will have a stable fixed point with steady state activity given by

$$\mathbf{v} = \mathbf{W} \cdot \mathbf{u} + \mathbf{M} \cdot \mathbf{v} \tag{14}$$

Defining $\mathbf{K} = (\mathbf{I} - \mathbf{M})^{-1}$, this has the solution $\mathbf{v} = \mathbf{K} \cdot \mathbf{W} \cdot \mathbf{u}$

If we now fix the recurrent weights and let the feedforward weights change according to a Hebbian learning rule, the time evolution of the weights is determined by equation 15 with $\mathbf{Q}$ defined as in section 1.

$$\tau_w \frac{d\mathbf{W}}{dt} = \langle \mathbf{vu} \rangle = \mathbf{K} \cdot \mathbf{W} \cdot \mathbf{Q} \tag{15}$$

We will use a discretized form of this equation with $\epsilon = 0.01$:

$$\mathbf{W}_{n+1} = \mathbf{W}_n + \epsilon \mathbf{KWQ} \tag{16}$$

We now consider a highly simplified model of ocular dominance including only a single direction along the cortex and a single point in the visual field. This gives rise to only two input activities u_R and u_L corresponding to the input from the right and left eyes. We include 512 output units in the model indexed with the label a which also specifies their location along the 1-dimensional cortex of length $L = 10$mm. We also assume the cortical interactions specified by $\mathbf{K}$ to be translationally invariant and impose periodic boundary conditions to avoid edge effects. Assuming that the right and left eye are statistically equivalent, we can now write the input correlation matrix as

$$\mathbf{Q} = \begin{bmatrix} \langle u_R u_R \rangle & \langle u_R u_L \rangle \\ \langle u_L u_R \rangle & \langle u_L u_L \rangle \end{bmatrix} = \begin{bmatrix} q_S & q_D \\ q_D & q_S \end{bmatrix}$$

Expanding the weight matrix $\mathbf{W}$ into its component vectors $\mathbf{w}_R$ and $\mathbf{w}_L$, we can consider the in-phase and out-of-phase combination of these vectors $\mathbf{w}_+ = \mathbf{w}_R + \mathbf{w}_L$ and $\mathbf{w}_- = \mathbf{w}_R - \mathbf{w}_L$ seperately. When expanding out equation 16, these evolve according to

$$\mathbf{w}_+^{n+1} = \epsilon(q_S + q_D)\mathbf{K} \cdot \mathbf{w}_+^n \qquad \mathbf{w}_-^{n+1} = \epsilon(q_S - q_D)\mathbf{K} \cdot \mathbf{w}_-^n \tag{17}$$

Using subtractive normalization for each pair of weights $w_L(a)$ and $w_R(a)$, we can leave $\mathbf{w}_+$ fixed while $\mathbf{w}_-$ changes. We therefore consider only $\mathbf{w}_-$ in the following and investigate how this weight vector changes over time.

Since the growth of $\mathbf{w}_-$ is proportional to $(q_S - q_D)$, these correlation parameters do not effect the long term behavior of the system but merely the timescale over which it changes provided that $q_S > q_D > 0$. We therefore arbitrarily let $q_S = 1$ and $q_D = 0.7$ for the remainder of this section.

In the present case, we implement subtractive normalization naively by performing the following update steps:

$$\mathbf{W}_{n+1} = \mathbf{W}_n + \epsilon\mathbf{K}\mathbf{W}\mathbf{Q} \tag{18}$$

$$\mathbf{W}_{n+1} = \mathbf{W}_{n+1} + 0.5(1 - \text{rowsum}(\mathbf{W}_{n+1})) \tag{19}$$

$$\mathbf{W}_{n+1}[\mathbf{W}_{n+1} < 0] = 0; \quad \mathbf{W}_{n+1}[\mathbf{W}_{n+1} > 1] = 1 \tag{20}$$

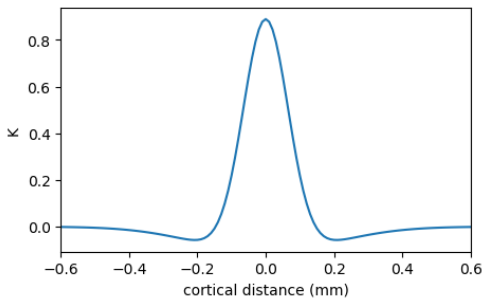
We intialize the weights such that $\mathbf{w}_L = 0.5 + \epsilon$ and $\mathbf{w}_R = 1 - \mathbf{w}_L$ where $\epsilon \sim \mathcal{N}(0, 0.01)$. This approach ensures that $\mathbf{w}_+ = 1$ at all times and implements subtractive normalization with thresholds of 0 and 1.

For this model, we generate cortical interactions as a function of intercortical distance according to equation 21 with $\sigma = 0.066$mm.

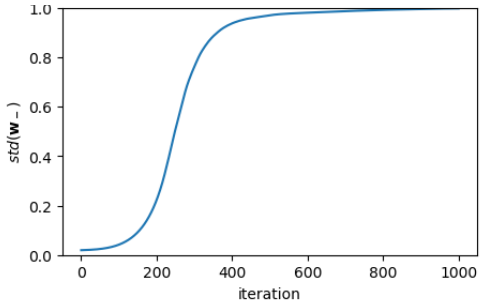
$$K_{aa'} = \exp\left(-\frac{(a-a')^2}{2\sigma^2}\right) - \frac{1}{9}\exp\left(-\frac{(a-a')^2}{18\sigma^2}\right) \tag{21}$$

The strength of this interaction has been plotted as a function of relative cortical position in figure 4a. We see that close-range interactions are strongly excitatory while long-range interactions are weakly inhibitory. This is what will result in an oscillatory pattern of ocular dominance.

Given this definition of cortical interactions, we can simulate the system of 512 cortical neurons by calculating the 512x512 interaction matrix $\mathbf{K}$ and implementing equations 18-20. In figure 4b, we plot the standard deviation of $\mathbf{w}_-$ which allows us to follow the progress of the simulation as $\mathbf{w}_-$ goes from having a standard deviation of 0 when all elements are ≈ 0 to having a standard deviation of 1 when the elements are ± 1. We see that the simulation has converged by 1000 timesteps and therefore run all simulations for 1000 timesteps unless otherwise noted.



(a) $K_{aa'}$ where a has been fixed a position 0 and a' varies from -0.6mm to 0.6mm.



(b) Standard deviation of $\mathbf{w}_-$ over time illustrating how $\mathbf{w}_-$ progresses from $\approx \mathbf{0}$ to a vector of ± 1 after 800-1000 iterations.

Figure 4

We can also extract the vector $\mathbf{w}_-$ at different points in the simulation and plot the vector as a 1-dimensional heatmap along the x-direction where individual neurons are coloured from white to black according their ocular dominance (figure 5).

From our previous considerations, we expect the long-term behavior of $\mathbf{w}_-$ to be dominated by the principal eigenvector of $\mathbf{K}$. In the case of periodic boundary conditions as imposed here, the eigenvectors of $\mathbf{K}$ are given by

$$e_a^\mu = \cos\left(\frac{2\pi\mu a}{512} - \phi\right) \tag{22}$$

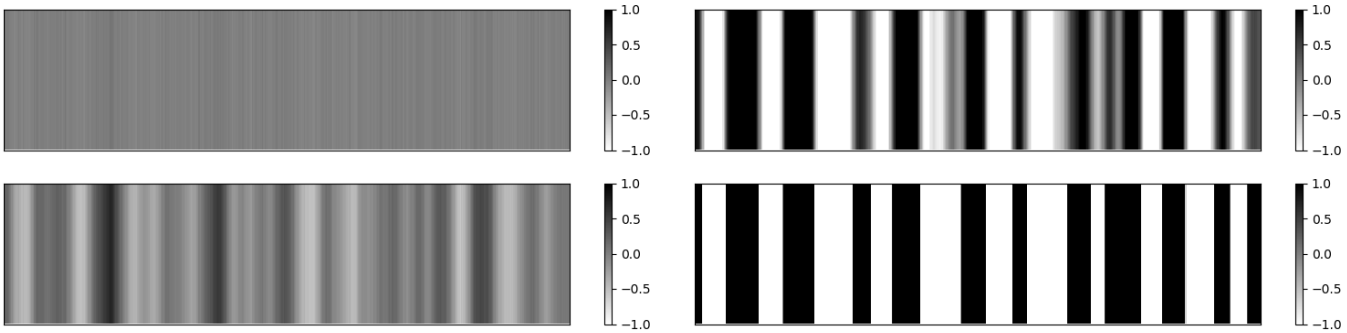


Figure 5: $\mathbf{w}_-$ as a heatmap after 50, 200, 300 and 1000 iterations. We see that ocular dominance becomes increasingly strong over time and that the short-range excitation with long-range inhibition drives the formation of an oscillatory pattern of dominance.

The eigenvalues $\mu = \dfrac{512 \cdot d \cdot k}{2\pi}$ take integer values given by the discrete fourier transform $\tilde{K}(\mu)$ of $\mathbf{K}$. Here, k is the spatial frequency of the ocular dominance columns and d is the separation between sites a and $a+1$; $d = \dfrac{10mm}{512}$. The principal eigenvector is thus the eigenfunction e_μ with μ corresponding to the maximum of $\tilde{K}(\mu)$.

To find the principal eigenvector of $\mathbf{K}$ and thus the expected long-term behavior of the system, we first find $\tilde{K}(k) = \int_{-\infty}^{\infty} K(x) e^{-2\pi i k x} dx$.

We know that the fourier transform is a linear operator and that the fourier transform of a Gaussian is given by

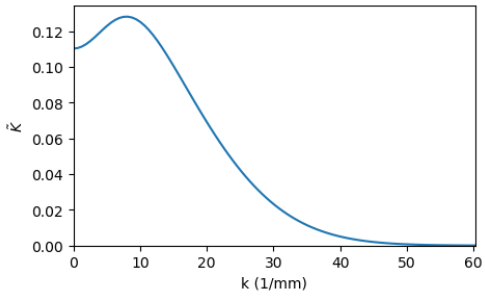
$$G(k) = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{\dfrac{-x^2}{2\sigma^2}} e^{-2\pi i k x} dx = e^{\dfrac{-k^2\sigma^2}{2}} \tag{23}$$

This allows us to easily calculate the required fourier transform

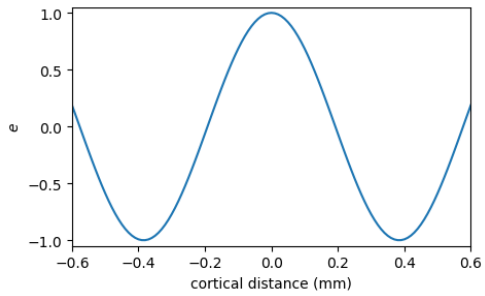
$$\tilde{K}(k) = \sqrt{2\pi\sigma^2} e^{\dfrac{-k^2\sigma^2}{2}} - \frac{1}{9}\sqrt{18\pi\sigma^2} e^{\dfrac{-9k^2\sigma^2}{2}} \tag{24}$$

We plot this in figure 6a and find that the maximum of $\tilde{K} = 0.128$ occurs at $k = 8.17$. We can now find the value of μ corresponding to this maximum $\tilde{K}$ as $\mu_{max} = \dfrac{512}{2\pi}\dfrac{8.17}{}\dfrac{10}{512} = 13$.

This allows us to find the principal eigenvector $e_{\mu_{max}}$ as a function of cortical distance and we plot this in figure 6b. We note that the pattern of inhibition and excitation follows that of K in figure 4a. Since $\mu_{max} = 13$, we expect to generate 13 periods of left-right dominance in our simulation, which is consistent with the results in figure 5.



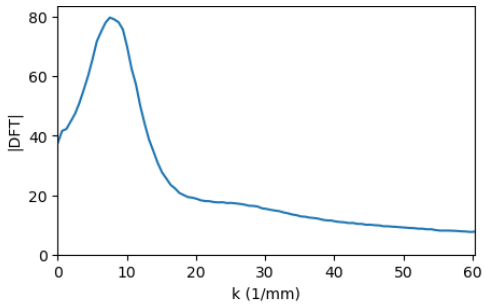
(a) Fourier transform of K as a function of spatial frequency k. This has a maximum of $k = 8.17$ corresponding to $\mu = 13$.



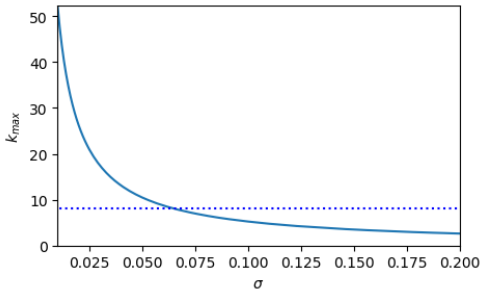
(b) Principal eigenvector of K capturing the pattern of short-range excitation and long-range inhibition.

Figure 6

We now repeat our simulation of the system 1000 times and calculate the magnitude of the discrete fourier transform (DFT) at each trial. We plot the mean of the DFTs in figure 7a and see that the major components of our equilibrium $\mathbf{w}_-$ vectors do indeed correspond to the values of k for which $\tilde{K}$ is highest in figure 6a.



(a) Mean magnitude of the discrete fourier transform of the steady state $\mathbf{w}_-$ vector across 1000 simulations.



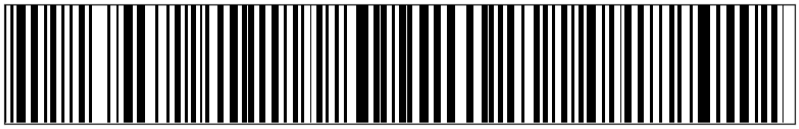
(b) Spatial frequency k that maximizes $\tilde{K}$ as a function of σ.

Figure 7

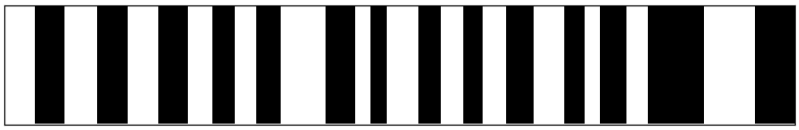
We find that the mean of the DFTs has a maximum of 79.7 at $\mu = 13$ which is consistent with our analytical considerations suggesting that the largest component of the long-time $\mathbf{w}_-$ vector should correspond to the principal eigenvector which does indeed have $\mu = 13$

The stripe width in this model is determined by the spatial range of excitatory and inhibitory interactions which in turn is specified by the scale parameter σ in equation 21. In order to decrease the stripe width in the model, we need to decrease the scale parameter σ which leads to an increase in the value of k that maximizes our discrete fourier transform and thus an increased frequency of the principal eigenvector of $\mathbf{K}$. In order to increase the stripe width, we instead increase σ. The relationship between σ and k has been investigated numerically, and the value of k that maximizes $\tilde{K}$ for a given σ has been plotted in figure 7b which shows a rapid increase in frequency as $\sigma \to 0$.

To validate these results, we run two additional simulations; one with $\sigma = 0.012$ for which we expect $k = 40.31$ and thus $\mu = 69$, and one with $\sigma = 0.174$ for which we expect $k = 3.01$ and $\mu = 5$. We plot the resulting steady state $\mathbf{w}_-$ vectors in figure 8 and see that we do indeed get increasingly wider stripes with increasing σ and that the number of full periods matches the expected value of μ.



(a) $\sigma = 0.012$ giving $k = 40.31$ and $\mu = 69$



(b) $\sigma = 0.066$ giving $k = 8.17$ and $\mu = 13$

(c) $\sigma = 0.174$ giving $k = 3.01$ and $\mu = 5$

Figure 8: Simulations of the ocular dominance system for different values of σ corresponding to different interaction ranges. Longer-range interactions lead to wider stripes.

We could of course also alter the cortical interactions in other ways, e.g. by altering the ratio of the two Gaussians from $1 : 1/9$ or by changing the relative standard deviations from $\sigma_2 = 3\sigma_1$, but these options are not explored further as they do not preserve the form of the interactions.

3 The elastic net

The Travelling salesman problem is an example of an NP complete problem, and there is thus no (current) algorithm that can solve it in polynomial time. As the number of nodes to be visited N increases, we are therefore forced to use various heuristic algorithms to try to approximate optimal solutions. One such algorithm dubbed the 'elastic net' was proposed by Durbin and Willshaw in a 1987 *letter to Nature*.

In this algorithm, there are N cities to be visited, and this is achieved by distorting an initial path of M points until it runs through all N cities in a way that locally minimizes the path length L. We denote the cities to be visited $\{x_i\}$ and the M points on the path travelled on $\{y_j\}$. At each timepoint the elastic net then updates each point on the path according to

$$\Delta y_j = \alpha \sum_i w_{ij}(x_i - y_j) + \beta K(y_{j+1} - 2y_j + y_{j-1}) \tag{25}$$

The first term serves to decrease the distance between a point on the path and a city, ensuring that all points on the path will eventually run through a city. The second term serves to bring the points as close to each other as possible, ensuring that the path length is minimized.

In equation 25, the weights w_{ij} are given by

$$w_{ij} = \frac{\phi(|x_i - y_j|, K)}{\sum_k \phi(|x_i - y_k|, K)}; \quad \phi(d, K) = \exp\left(\frac{-d^2}{2K^2}\right) \tag{26}$$

This implementation of the elastic net corresponds to minimizing an energy function

$$E = -\alpha K(\sum_i \ln \sum_j \phi(|x_i - y_j|, K)) + \beta \sum_j |y_{j+1} - y_j|^2 \tag{27}$$

Where again the first term penalizes when the path is far from a city and the second term penalizes a long path. Our update step from equation 25 then has the property $\Delta y_j = -K\frac{\partial E}{\partial y_j}$.

We are thus carrying out gradient descent on equation 27 with adaptive learning rate K, ensuring that we will eventually arrive at a local minimum of the energy.

Over the course of the optimization, Durbin and Willshaw reduce K by 1% every 25 iterations from $K = 0.2$ to $K = 0.01$ and fix $M = 2.5N$. However, given the advances in computing power since 1987, we can explore the effect of K and M on the performance of the algorithm more thoroughly.

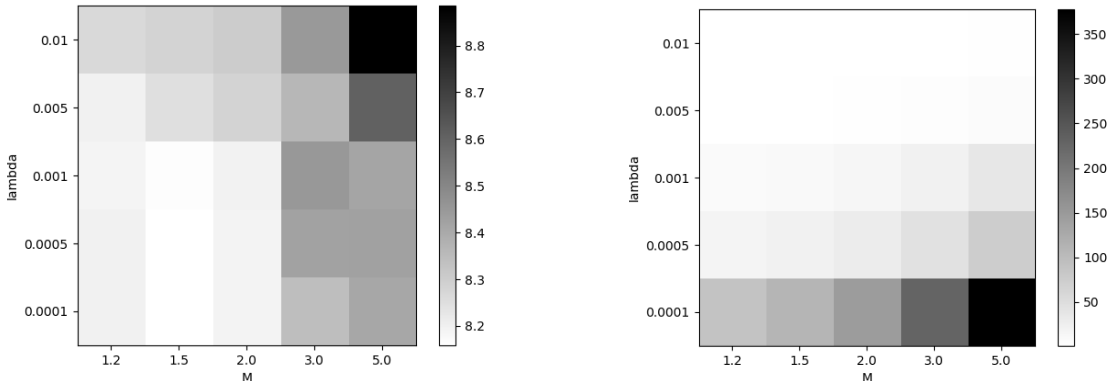
For the remainder of this section we set $\alpha = 0.2$ and $\beta = 2.0$ as in Durbin and Willshaw, and we choose points on our initial path to be evenly spread out at a distance $0.1 + \text{unif}(-0.001, 0.001)$ from the centroid of the cities. Cities are generated as a grid of 100 randomly distributed points in a 1x1 square.

We let K decay exponentially according to $K = 0.2 * \exp(-\lambda n)$ where n is the number of timesteps, and a simulation is considered finished when $K < 0.001$. We can now vary lambda and the number of points M on our path to optimize the elastic net. The result of a coarse-grained parameter search is given in figure 9.

We see from figure 9a that in contrast to the parameters used by Durbin and Willshaw, we achieve the best performance with a relatively small value of M=1.5N since small values of M tend to reduce the popensity of the system to get caught in higher-energy local minima compared to $M > 2N$. We also note that there is an increase in performance with decreasing lambda as expected, but this effect is neglible for $\lambda < 0.0005$.

From figure 9b we see that the time taken for a simulation is approximately linear in both λ and M. Considering this data together with figure 9a, we fix M=1.5 and $\lambda = 0.0005$ for the remainder of our simulations. A more finegrained optimization could be carried out, but this was not deemed particularly informative since the details of the optimization will depend upon the specific grid being investigated. However, the above analysis was repeated with two additional random grids and similar patterns were observed.

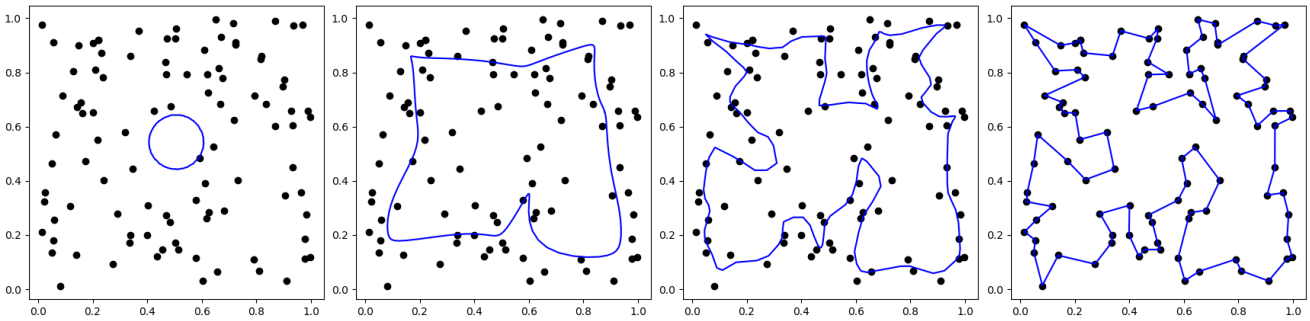
We can now investigate how the path develops as K decreases for our optimum parameters of M=1.5 and $\lambda = 0.0005$, with four timepoints during the simulation shown in figure 10. We see that the path initially expands relatively uniformly to decrease the distance from the far-away cities to the path. The path then locally distorts as it moves closer to every individual point. This is finally achieved at K=0.001 where the path has converged and the salesman visits every city. When M gets too close to N, the algorithm occasionally converges to a local minimum where the path does not visit every city. For $M \geq 1.5N$ this has not been observed.



(a) Length of final path

(b) Time to convergence (seconds)

Figure 9: Performance (left) and computational time (right) for the elastic net approach to the travelling salesman problem as a function of decay rate λ and number of points on the path M. All simulations were run on the same set of 100 cities with remaining parameters as specified in the main text.



(a) K=0.2

(b) K=0.11

(c) K=0.04

(d) K=0.001

Figure 10: Timecourse of the travelling salesman simulation with M=1.5 and $\lambda = 0.0005$

At a first glance, this looks like a reasonable route as there are no obvious detours or crossovers. However, to get a better idea of how efficient the elastic net method is, we can compare it to the well established travelling salesman method of simulated annealing. For this comparison, we implement simulated annealing as described in *Stochastic Processes* (Chang 2007).

We can consider a given TSP route (order of cities) to be a point in a set $\mathcal{S}$ of $\frac{(n-1)!}{2}$ possible routes. Our aim is to find a route that minimizes the total length L by moving between neighboring points on $\mathcal{S}$. Here, we define 'neighboring points' as routes that can be interconverted by reversing the part of the path between two cities.

In the simulated annealing algorithm, one iteration involves moving uniformly at random to a neighboring point in $\mathcal{S}$ and calculating the length $L1$ of the new route obtained. If $L1 < L0$, we accept this new route as our current route. If $L1 > L0$, we accept the new route with probability $p = \exp\left(-\frac{L1 - L0}{T}\right)$. Otherwise we retain the old route.

This finite probability of accepting a worse route allows us to move out of local minima and thus to approach the global minimum with a higher probability than a simple gradient descent algorithm. This is similar to e.g. the annealing process in glass (from which the algorithm has its name), or protein folding in a cell where the finite temperature allows for stochastic movement out of local minima in conformational space.

Over the course of the simulation, we decrease the tempeature according to $T_n = \frac{T_0}{ln(n)}$. Here n is the number of iterations. The decrease in temperature makes it increasingly unlikely that we will move to a worse route, and in the limit of $T \to 0$, the algorithm converges to a local minimum as there is no thermal energy in the system to drive it to a longer path.

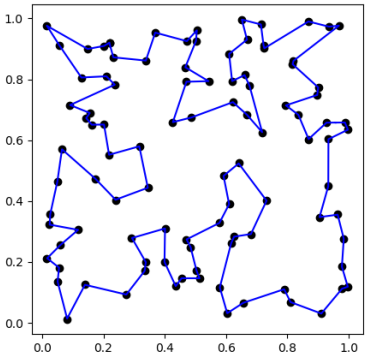
Setting $T_0 = 10$ and running 10^7 iterations has been found empirically to give good results on a timescale similar to the one observed for the elastic net ($\approx$ 20 seconds). To compare the two methods, we use the map of 100 cities

considered above as well as two new randomly generated maps. We then quantify the optimum route generated by the elastic net in terms of both route length and time to convergence. We compare this to 30 trials of simulated annealing since the simulated annealing process is stochastic and results thus vary per trial. A summary of the results is given in table 1.

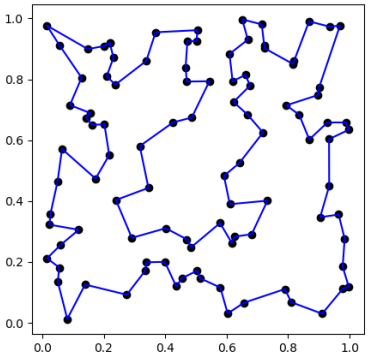
	Map 1	Map 2	Map 3
Elastic net	8.159	7.771	7.815
Time (s)	23.32	23.819	24.91
Simulated annealing	7.956 (0.047)	7.768 (0.052)	7.633 (0.075)
Time (s)	19.06 (1.33)	19.47 (1.97)	18.47 (1.12)

Table 1: Performance and computational time for the elastic net and simulated annealing across three different maps of 100 randomly generated cities. For simulated annealing, results are reported as mean (std).

We see that simulated annealing consistently outperforms the elastic net, both in terms of the converged route and the time taken to find this route. This can be seen qualitatively in figure 11.



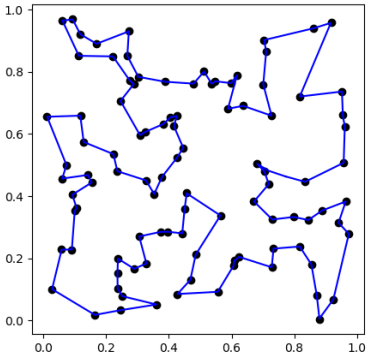
(a) Elastic net (L=8.16)



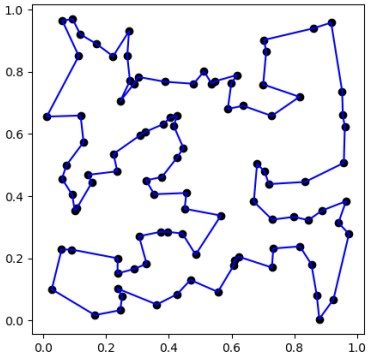
(b) Simulated annealing (L = 7.92)

Figure 11: Optimum routes for map 1 found by the elastic net method and simulated annealing. Simulated annealing finds a shorter route than the elastic net, and this route involves only visiting the inner region of the map once.

However, for map 2 simulated annealing only slightly outperforms the elastic net on average and performs worse in almost half the trials, raising the question of whether there are any profound insights to be gained from this difference between maps. For map 1, we find that the optimum route from simulated annealing appears very asymmetric. It is thus less likely to be converged upon by the elastic net, the dynamics of which lead to an initial expansion of the route followed by the formation of local invaginations (figure 10). For map 2, on the contrary, the optimum route from simulated annealing appears more symmetrical which might explain why the elastic net comes close to the performance of simulated annealing in this case (figure 12).



(a) Elastic net (L = 7.77)



(b) Simulated annealing (L = 7.69)

Figure 12: For map 2, the optimum solution found by simulated annealing is more symmetrical, and the elastic net comes close to the performance of simulated annealing.

Simulated annealing is thus a superior method to the elastic net in terms of solving the travelling salesman problem, but we see that the elastic net still produces reasonable routes that are comparable in length to those generated by simulated annealing albeit generally more symmetric. The elastic net methodology is therefore still interesting due to its dynamics and general applicability to problems involving mappings between different topologies.

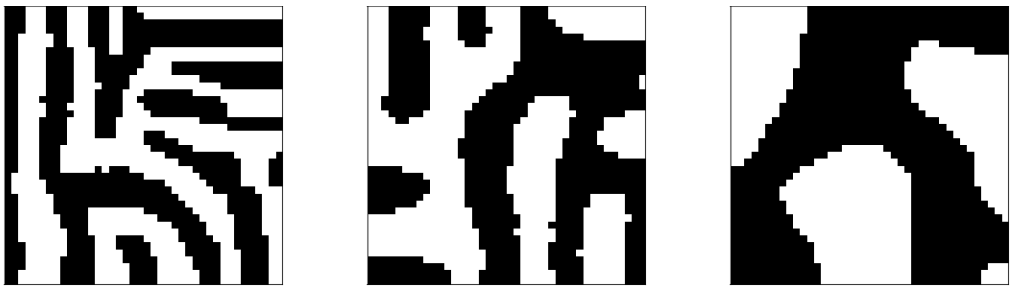
As an example, we use the elastic net to model ocular dominance in two dimensions, inspired by Goodhill and Willshaw (1989). We imagine the 'tour' as being a plane representing a region of the visual cortex where each point has elastic connections to its four neighboring points. The retinal inputs are represented by two horizontal planes corresponding to the left and right eyes respectively with each point being the equivalent of a city in the TSP. We now index our cortical plane by i, j and the retinal positions by a, b. This gives rise to a modified update step:

$$\Delta y_{ij} = \alpha \sum_{a,b} w_{ab,ij}(x_{ab} - y_{ij}) + \beta K(y_{i\,j+1} + y_{i+1\,j} - 4y_{i\,j} + y_{i\,j-1} + y_{i-1\,j}) \tag{28}$$

The weights $w_{ab,ij}$ are given by Gaussians of Euclidean distances as before. The degree of correlation between left and right eyes are defined by the distance $2l$ between the two ocular planes and the inter-point separation $2d$ determines the separation of photoreceptors.

The stripe width of the emerging pattern is governed by the ratio l/d. As l/d increases, cortical-cortical correlation becomes stronger relative to retinal correlation, leading to broader stripes. As l/d decreases, the opposite trend is observed.

This is illustrated in figure 13 where we reproduce the result of Goodhill and Willshaw (1989) showing the effect of l/d on stripe width. We fix $2d = 0.05$ and let $2l$ be equal to 0.10 (figure 13a), 0.15 (figure 13b) and 0.20 (figure 13c) leading to successive increases in stripe width. For these simulations we fix $N = 20$ and $M = 40$ where N and M are the squareroots of the number of retinal and cortical points respectively.



(a) $2l = 0.10$; $2d = 0.05$ (b) $2l = 0.15$; $2d = 0.05$ (c) $2l = 0.20$; $2d = 0.05$

Figure 13: Ocular dominance maps resulting from a two-dimensional elastic net simulation. Individual squares (cortical neurons) are coloured according to whether they are most closely associated with the right (white) or left (black) retina. Stripe width increases with l/d.

For the purpose of these simulations, cortical points were initialized uniformly at random in the xy plane corresponding to random initial retinal-cortical connectivity. The z locations of the points were generated uniformly at random between ± 0.01 corresponding to balanced but noisy ocular input.

Interestingly, we see from figure 14 that in addition to the emergence of ocular dominance, this simulation also leads to the emergence of visual cortical maps as has been observed experimentally. From our initial random connectivity, we see that at the end of the simulation, cortical neurons with large y-values receive input from retinal neurons with small x-values (white) and vice versa. Similarly, neurons with small x-values in the cortex receive input from retinal neurons with small y-values and vice versa (not shown). This swap of the x and y axes between the retinal and cortical coordinate systems may at first seem surprising, but since our energy landscape has D_{4h} symmetry, this is entirely equivalent to an $x \to x$ and $y \to y$ mapping.

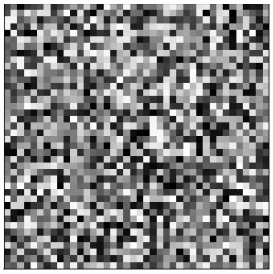
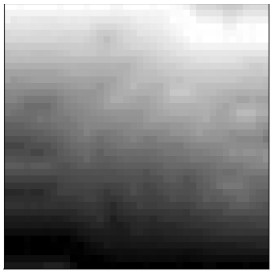



Figure 14: Retinal-to-cortical mapping in the retinal x direction at the beginning (left) and end (right) of a simulation with $2l = 0.15$ and $2d = 0.05$. x and y axes correspond to locations in the cortex. Colours indicate retinal connectivity (i.e. location in retinal space) with white and black regions corresponding to cortical neurons with input from retinal neurons with low and high x-values respectively. We see that neurons self-organize from the random initial connectivity to generate a cortical map.

4 Cart pole balancing problem

The Cart Pole Balancing Problem is a commonly used task in machine learning and computational neuroscience to assess the performance and adaptability of a network. In the following, we solve this task using a network described by Barto et al. (1983). This network consists of only two elements: an *Associate Search Element* (ASE) which picks a decision based on a reward signal and eligibility trace, and an *Adaptive Critic Element* (ACE) which provides the ASE with a predicted reward over continuous time rather than a discrete reward at the end of a trial.

The state of the cartpole system is defined by a vector $(x, \theta, \dot{x}, \dot{\theta})$ where x is the position of the cart on a one-dimensional track with $x \in [-2.4, 2.4]$, θ is the angle of the pole from vertical and dots represent time derivatives. A trial is considered to have failed when $|x| \geq 2.4$ or $|\theta| \geq 12°$. We divide the state space into 162 distinct regions defined by
$x \in [-2.4, -0.8], [-0.8, 0.8], [0.8, 2.4]$ (m)
$\theta \in [-12, -6], [-6, -1], [-1, 0], [0, 1], [1, 6], [6, 12]$ (°)
$\dot{x} \in [-\infty, -0.5], [-0.5, 0.5], [0.5, \infty]$ (m/s)
$\dot{\theta} \in [-\infty, -50], [-50, 50], [50, \infty]$ (°/s)

At any time t, the system will be in a given state $state(t)$ and it receives an impulse $y(t)$ either to the right or to the left depending on the relative predicted reward for right- and left-wards impulses. We therefore map a given state vector to a binary 162-dimensional vector x that has all elements equal to zero except for the element corresponding to the current state of the system, and let the impulse at time t be given by equation 29.

$$y(t) = \text{sign}[\sum_{i}^{162} w_i(t)x_i(t) + \mathcal{N}(0, \sigma^2)] = \text{sign}[w_{state(t)}(t) + \mathcal{N}(0, \sigma^2)] \tag{29}$$

Here $x_i(t)$ is 1 if the system is in state i at time t and zero otherwise, and $w_{state(t)}(t)$ is the weight at time t corresponding to the state of the system at time t. The Gaussian noise both simulates noise in real neural systems and allows our cart pole balancing system to explore the available state space.

For this ASE we update the weights according to

$$w_i(t+1) = w_i(t) + \alpha r(t) e_i(t) \tag{30}$$

This is a three-factor learning rule where α is the learning rate, $r(t)$ is the external reward at time t and $e_i(t)$ is the eligibility trace of unit i at time t. For the pole balancing problem, $r = 0$ throughout a trial and becomes -1 when failure occurs. We update the eligiblity traces according to

$$e_i(t+1) = \delta e_i(t) + (1 - \delta) y(t) x_i(t) \tag{31}$$

where δ determines the trace decay rate and x, y are as defined above.

The ASE on its own does not perform particularly well given the sparse error signal which only occurs after a long sequence of events. Performance is better if we include an ACE which provides an internal reinforcement signal at all times. This internal reinforcement signal is given by

$$\hat{r}(t) = r(t) + \gamma p(t) - p(t-1) \tag{32}$$

Here, γ leads to the predicted reward tending towards 0 in the case of prolonged periods without external reinforcement. $p(t)$ is a prediction of eventual reinforcement, given by

$$p(t) = \sum_{i}^{162} v_i(t)x_i(t) = v_{state(t)}(t) \tag{33}$$

This requires us to update the weights v_i such that they remain an accurate predictor of reward. This is achieved using the learning rule

$$v_i(t+1) = v_i(t) + \beta[r(t) + \gamma p(t) - p(t-1)]\bar{x}_i(t) \tag{34}$$

Here, β determines the learning rate and $r(t)$ is the external reinforcement signal as above. $\bar{x}_i(t)$ is similar to the eligibility trace $e_i(t)$ for the ASE and is updated according to

$$\bar{x}_i(t+1) = \lambda\bar{x}_i(t) + (1-\lambda)x_i(t) \tag{35}$$

Where λ is the trace decay rate. This is equivalent to the update rule for $e_i(t)$ in the case where $y_i(t) = 1$.

We initialize all $e_i, w_i, v_i, \bar{x}_i = 0$ at time $t = 0$. We intialize the cart at x=0 and the pole at an angle of 0 degrees. We then integrate the above equations using Euler integration with a timestep of 0.02 as in Barto et al. and similarly let the parameters of the system be given by $\alpha = 1000$, $\beta = 0.5$, $\delta = 0.9$, $\gamma = 0.95$, $\lambda = 0.80$, $\sigma = 0.01$.

Using these parameters, the system learns the balancing task in 20-80 trials and we show the result of the first 1000 seconds of simulated time from an example simulation in figure 15.

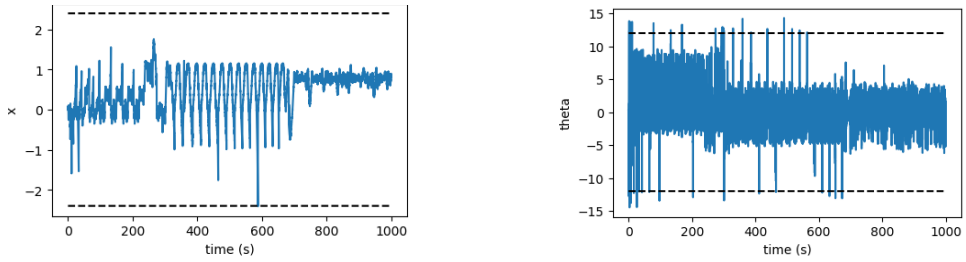
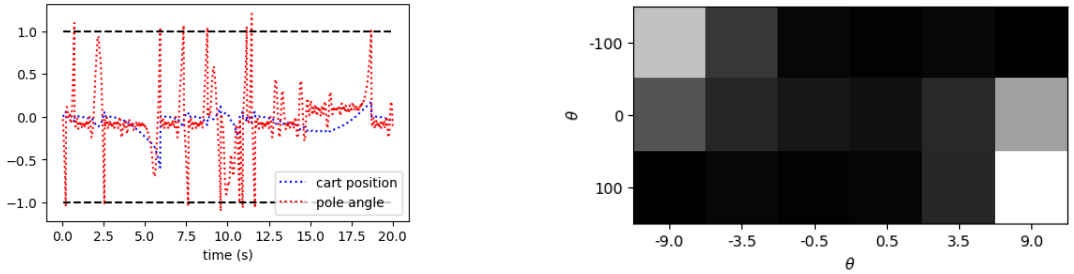


Figure 15: Cart pole balancing simulation showing x (left) and theta (right) over 1000 seconds of simulated time.

We see that the system more commonly fails due to the pole falling than due to the cart exiting the arena. However, these two variables are correlated so we expect learning in one parameter to also help performance in the other. This is illustrated in figure 16a where we see that in many cases where the poll falls, the cart is also gaining momentum.

Over the course of the simulation we expect the system to learn that states with high θ and high $\dot{\theta}$ are associated with low eventual reward as they will likely lead to system failure, and similarly for states with very negative θ and $\dot{\theta}$. To verify this, we plot the value of v as a function of θ and $\dot{\theta}$ averaged over x and $\dot{x}$ in figure 16b after 10,000 seconds of simulated time.



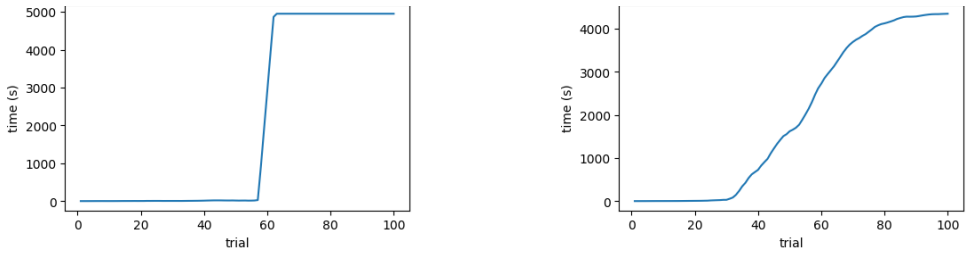
(a) Cart position x and pole angle θ for the first 20 seconds of a simulation. y-axis is scaled by the maximum values.

(b) v averaged over x and $\dot{x}$ as an indication of expected reward for different combinations of θ and $\dot{\theta}$. Black indicates higher expected reward.

Figure 16

We see that these extreme states do indeed have a low predicted reward and that v has a 'ridge' of states with high predicted reward running from the lower left to the upper right corner.

To investigate the rate of learning, we follow the example of Barto et al. and run the network for 500,000 iterations corresponding to 100,000 seconds of simulated time. We then quantify the length of each trial with a trial terminating once the system fails. After each trial, we reset all eligibility traces e_i to 0. We then plot the trial length against trial number for a single simulation (17a) or as an average over 500 simulations (17b).

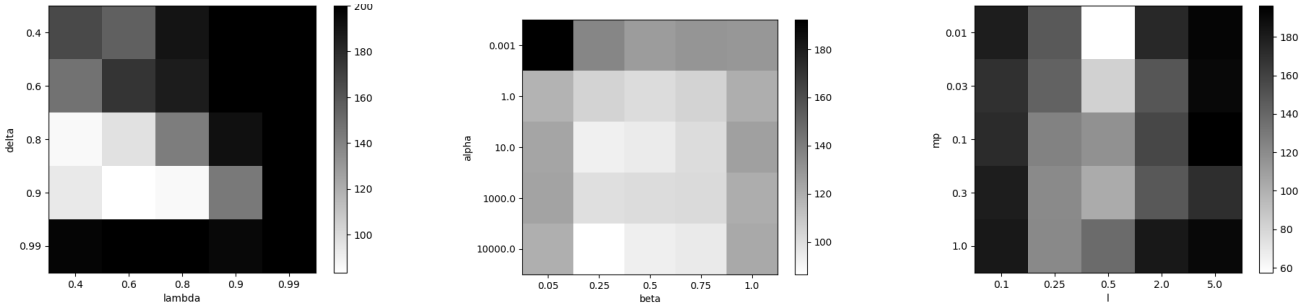


(a) Performance for an example trial.

(b) Mean performance over 500 trials.

Figure 17: Plots of trial duration (simulated time) against trial number. We see that for a single trial, there is a very steep increase in trial duration after the final failure, and over 500 trials this averages out to approximately linear learning between trial 30 and 70. By 100 trials, there are generally no more failures.

Having investigated the rate of learning for the parameters used in Barto et al., we proceed to investigate the effect of some of the key parameters on the learning rate. For the purpose of these investigations, we consider the system to have learned when the mean learning curve of the system over 100 simulations exceeds 10% of the total simulation time; i.e. when the mean trial length exceeds 1000 seconds of simulated time. We then average this performance over 10 sets of simulations and calculate this metric for a range of different parameters in figure 18.



(a) Performance for different eligibility trace decay rates δ and λ

(b) Performance for different ASE and ACE learning rates α and β

(c) Performance for different pole masses (mp) and lengths (l).

Figure 18: Learning performance as a function of different sets of parameters. The value given is the number of trials needed for the mean trial length to exceed 1000 seconds of simulated time. Lower values indicate better performance. The simulations were capped at an upper limit of 200 trials and all other parameters were fixed at their default values.

We see from figure 18a that performance strongly depends on the eligibility trace decay rates δ and λ with optimum performance observed for $\delta = 0.9$ and $\lambda = 0.6 - 0.8$. This is very similar to the parameters used by Barto et al. Performance drops at very low decay rates as this prevents the system from correctly penalizing recent states that lead to failure, and performance drops at high decay rates as early states are penalized even if they are far removed from the failure of the system, leading to less differentiation between 'good' and 'bad' states.

On the contrary, the learning rates α and β have surprisingly little effect on the performance of the system with a wide range of parameter values giving performances in a range of 80-140 trials (figure 18b). At very low α or β, the system fails to learn over the course of the simulation and performance deteriorates due to a relative increase in the noise term. At high β, early trials continue to dominate the ACE throughout the simulation and performance also gets worse. However, at high α the behavior of the system saturates without a drop in performance since we are just scaling the weights linearly corresponding to a change of units.

In figure 18c we see that the performance is highly dependent on pole length (l) but less so on pole mass (mp). The task becomes easier as the pole becomes lighter since this leads to slower dynamics and thus less frequent pole drops. The task becomes harder when the pole becomes too short since $|\dot{\theta}|$ increases with decreasing l ($\dot{\theta} \propto 1/l$). It also becomes harder as the pole gets too long since $|\dot{x}|$ increases with increasing l. $l = 0.5$ represents a balance between

these two effects.

Having investigated the performance of this simple model, we can now add a second layer to the network as described by Anderson (1987). In this case, the input $\mathbf{x}$ is the full state vector $[x, \theta, \dot{x}, \dot{\theta}]$ scaled as described in Anderson (1987), with an additional bias unit that has a constant value of 0.5. Since the task is thus quite different from the task in Barto et al., we also perform an internal comparison with a single-layer Anderson network.

We denote the activities of the hidden ASE layer z_i with weights from the input layer given by the matrix $\mathbf{D}$ such that

$$z_i(t) = g(\sum_{j=1} D_{ij}(t)x_j(t)) \tag{36}$$

Where $g(s) = \dfrac{1}{1+e^{-s}}$. This allows us to calculate an output probability $P(t) = g(\sum_{i=1} w_i(t)x_i(t) + \sum_{i=1} f_i(t)z_i(t))$ where f_i are the weights from the hidden layer to the output layer (equivalent to w_i for the input layer). We then have $y(t) = 1$ with probability $P(t)$ and $y(t) = -1$ with probability $1 - P(t)$. Our ACE is also expanded by a hidden layer q with connectivity matrix $\mathbf{A}$ from the input layer, giving activities

$$q_i(t1, t2) = g(\sum_{j=1} A_{ij}(t1)x_j(t2)) \tag{37}$$

The output of the ACE is given by

$$p(t1, t2) = \sum_{i=1} v_i(t1)x_i(t2) + \sum_{i=1} c_i(t1)q_i(t2) \tag{38}$$

This gives us an expected reward $\hat{r} = 0$ if the system is in a start state, $\hat{r} = r - p(t, t)$ if the system is in a failure state and $\hat{r} = r + \gamma p(t, t+1) - p(t, t)$ otherwise. We then update the ASE weights according to

$$w_i(t+1) = w_i(t) + \alpha\hat{r}(\max(y(t), 0) - P(t))x_i(t) \tag{39}$$

$$f_i(t+1) = f_i(t) + \alpha\hat{r}(\max(y(t), 0) - P(t))z_i(t) \tag{40}$$

$$D_{ij}(t+1) = D_{ij}(t) + \alpha\hat{r}z_i(t)(1 - z_i(t))\text{sign}(f_i(t))(\max(y(t), 0) - P(t))x_j(t) \tag{41}$$

We update the ACE weights according to

$$v_i(t+1) = v_i(t) + \beta\hat{r}x_i(t) \tag{42}$$

$$c_i(t+1) = c_i(t) + \beta\hat{r}q_i(t, t) \tag{43}$$

$$A_{ij}(t+1) = A_{ij}(t) + \beta\hat{r}q_i(t, t)(1 - q_i(t, t))\text{sign}(c_i(t))x_j(t) \tag{44}$$

Figure 19 shows that this two-layer implementation does indeed learn the task in roughly 2000 trials while the single-layer performance saturates at a performance of 20 seconds of simulated time after 500 trials. Interestingly, the single-layer network outperforms the two-layer network for the first 1000 trials until the two-layer network forms 'helpful features' as described by Anderson.

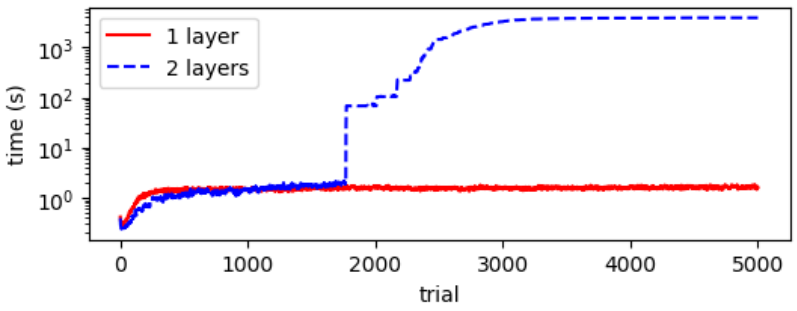


Figure 19: Performance of the Anderson network using a network with one or two layers. x-axis indicates trial number and y-axis indicates trial length in simulated time.

Finally we note that learning in the two-layer Anderson network is still an order of magnitude slower than in the Barto implementation since Barto et al.'s network only needs to consider a discrete set of 162 possible states while the Anderson network has to generalize over a continous set of states.

Appendix

```
#Code for unsupervised Hebbian learning tasks

using PyCall, PyPlot, LinearAlgebra, Distributions,
    Random, DelimitedFiles, LaTeXStrings, StatsBase, Statistics, MultivariateStats
Random.seed!(06032039) #random seed for reproducibility

function norm2d(;n=500, m=0, sd=1, rho=-0.7, slope=1)
    #same mean and sd for both distributions.
    #rho defines covariance, slope defines slope.
    #return data for a 2D Gaussian distribution
    d1 = Normal(m, sd)
    x = rand(d1, n)
    d2 = Normal(m-rho*m, sqrt(1-rho^2))
    y = rand(d2, n)+rho*slope*x
    #plot input data
    figure()
    plot(x,y, "bo")
    xlim(min(minimum(x), minimum(y)), max(maximum(x), maximum(y)))
    ylim(min(minimum(x), minimum(y)), max(maximum(x), maximum(y)))
    savefig("testfig.png")
    show()
    close()
    println(mean(x)," ", mean(y))
    println(cov(x, y))
    return [x';y']
end


function update_additive(w,C,n,dt)
    #performs an update step for hebbian learning with subtractive normalization
    m = mean(w)
    finished = false
    neww = w
    neww = w + dt * (C * w .* n - ((w'*C*n)/sum(n))*n)
    ____for i in 1:length(w)
    ________if neww[i] < 0
    ____________#println("zeroing weight ", i)
    ____________neww[i] = 0 #set weight to zero
    ____________n[i] = 0 #don't alter this weight anymore
            neww = neww*m/mean(neww) #ensure sum of weights is constant
        end
    end
    return neww, n
end

function simulate_system(input;tstop=10, mult = true, corr=true, dt=0.01, Print=true,
                w=[0.001,-0.002], alpha=1, thresh = 10^(-6), stat=false)
    #performs a full simulation using either multiplicative (mult=true)
    #or subtractive (mult=false) normalization
    #corr=true for correlation-based training, false for covariance-based
    C = zeros(2,2); n = ones(2)
    if !corr #subtract mean of input if we use a covariance based learning rule
        input = input .- mean(input,dims=2)
    end
    for i=1:2
        for j=1:2 #fill in correlation/covariance matrix
```

```
            C[i,j] = (input[i,:]' * input[j,:])/size(input)[2]
        end
    end
    Print && println(C)
    M = fit(PCA, input) #perform PCA
    #stat=true to initialize at stationary point
    if stat w = [1, (C[1,1]-C[1,2])/(C[2,2]-C[1,2])]; w = w./sum(w) end
    Print && println(w)
    N = Int(tstop/dt)+1

    ws = w
    w0 = w
    err = thresh+1

    while err > thresh #thresh is weight change for convergence
        if mult #multiplicative normalization
            w = w + dt*(C*w - alpha*(w'*C*w)*w)
        else
            w, n = update_additive(w,C,n,dt) #additive
        end
        ws = hcat(ws, w) #store weights
        err = norm(w-w0)
        w0 = w
    end
    Print && println("final vec: ", ws[:,end])
    Print && println("PCs: ", M.proj[:,1], " ", M.proj[:,2])
    return ws
end

function plot_dat_weights(dat, weights, name; m=0)
    #function for plotting data and weights. m is mean of data. assume equal means

    figure(figsize = (4,4))
    plot(dat[1,:], dat[2,:], "bo", MarkerSize = 1) #plot data
    rat = weights[2,end]/weights[1,end] #slope of weight vector
    if rat < 0 #shift line so it runs through data
        ys = [-10;10]*rat.+2*m
    else
        ys = [-10;10]*rat
    end
    plot([-10;10], ys, "r:") #plot direction of final weight vector
    xlabel(L"u_1, w_1")
    ylabel(L"u_2, w_2")
    xlim(m-3, m+3)
    ylim(m-3, m+3)
    savefig("figures/"*name*".png")
    close()

    norms = [] #Plot weight vector norms vs time
    for i=1:size(weights)[2]
        norms = [norms;norm(weights[:,i])]
    end
    figure(figsize = (4,3))
    plot(norms, "b-")
    xlabel("iteration")
    ylabel("|w|")
    xlim(0, length(norms))
```

```
    ylim(0,1)
    savefig("figures/"*name*"_vec.png", bbox_inches="tight")
    close()
end

function make_weight_fig(;basename="mul", rho=-0.7, mult=true)
    #function for reproducing TN figure
    dat1 = norm2d(n=1000, m=0, rho=rho) #generate data
    dat2 = norm2d(n=1000, m=3, rho=rho)
    dat3 = norm2d(n=1000, m=3, rho=rho)
    corrs = [true, true, false]
    ms = [0;3;3]
    #set initial weights
    if mult weights = [0.001, 0.001] else weights=[0.4, 0.6] end
    for (index, dat) in enumerate([dat1, dat2, dat3])
        #run simulation and plot result for each dataset
        ws = simulate_system(dat, corr=corrs[index], mult=mult, w=weights)
        plot_dat_weights(dat, ws, basename*"_2d_sim"*string(index), m=ms[index])
    end
end

function repeat_sub(;n = 100, name="test", rho=0.7, slope=1)
    #repeatedly run subtractive normalization for data with
    #orrelation rho and slope slope. Plot histogram of results
    thetas = [] #angles from 1st PC
    for i = 1:n
        dat = norm2d(n=1000, m=0, rho=rho, slope=slope) #generate data
        winit = rand(2); winit /= sum(winit) #random initial vectors
        ws = simulate_system(dat, mult=false, w=winit, corr=false,
            stat=false, Print=false)
        w = ws[:,end] #get final vector
        M = fit(PCA, dat)
        pc1 = M.proj[:,1]
        theta = min(acos(pc1' * w / (norm(pc1)*norm(w))),
                    acos(-pc1' * w / (norm(pc1)*norm(w)))) #compare to 1st PC
        thetas = [thetas; theta]
    end
    #plot histogram of results
    figure(figsize = (4,3))
    PyPlot.plt[:hist](thetas)
    xlabel("theta")
    ylabel("frequency")
    savefig("figures/"*name*"_hist.png", bbox_inches="tight")
    close()
end

make_weight_fig(rho=-0.7, basename="mult", mult = true) #reproduce TN figure
make_weight_fig(rho=-0.7, basename="sub", mult = false) #try subtractive norm

dat = norm2d(n=1000, m=0, rho=0.7, slope=1) #initialize at stationary point
ws = simulate_system(dat, mult=false, w=[0.5,0.5], corr=false, stat=true)
plot_dat_weights(dat, ws, "stationary"; m=0)

dat = norm2d(n=1000, m=0, rho=0.995, slope=0) #1st PC is [1,0]
ws = simulate_system(dat, mult=false, w=[0.5,0.5], corr=false, stat=false)
plot_dat_weights(dat, ws, "1_0"; m=0)
```

```
repeat_sub(;n = 100, name="slope_1", rho=0.7, slope=1) #histogram for slope of 1
repeat_sub(;n = 100, name="slope_02", rho=0.7, slope=0.2) #histogram for slope 0.2

#Code for simulatign and analyzing occular dominance

using PyCall, PyPlot, LinearAlgebra, Distributions, FFTW,
    Random, DelimitedFiles, LaTeXStrings, StatsBase, Statistics, MultivariateStats

function plot_occ(wm; xlab="", ylab="", filename="default", Title="default")
        #plots the degree of occular dominance as a 1d heatmap
        Wm = zeros(1,512)
        Wm[1,:] = wm #1x512 array
        figure(figsize = (10,1.5))
        imshow(Wm, cmap=ColorMap("gray_r"), vmin = -1, vmax = +1, aspect="auto")
        print("plotted")
        xticks([], [])
        yticks([], [])
        savefig(filename, bbox_inches = "tight")
        close()
end

function get_int(d1, d2; n = 512, sigma = 0.066, L = 10)
    #return K(|d1-d2|) as specified in the question
    Kij = exp( -( d1 - d2 )^2/(2*sigma^2) ) -
            1/9 * exp( -( d1 - d2 )^2/(18*sigma^2) )
    return Kij
end

function get_trans(k; n = 512, sigma = 0.066, L = 10)
    #for a given value of k, returns the fourier transform of K for this k
    Kij = sqrt(2*pi*sigma^2)*exp(-k^2*sigma^2/2) -
        1/9 * sqrt(18*pi*sigma^2) * exp(-9*k^2*sigma^2/2)
    return Kij
end

function plot_Ktildes()
    #plots the fourier transform of K as a function of k for a range
    #of mu values
    mus = 0:1:96
    ks = [2*pi/10*mu for mu in mus]
    Ks = [get_trans(k) for k in ks]
    figure(figsize = (5,3))
    plot(ks, Ks)
    xlabel("k (1/mm)")
    ylabel(L"\tilde K")
    xlim(minimum(ks), maximum(ks))
    ylim(0, maximum(Ks*1.05))
    savefig("figures/plot_Ktilde.png", bbox_inches = "tight")
    close()
    return ks, Ks
end

function plot_Ks()
    #plots K(0, d) for a range of intercortical distances
    ds = -0.6:0.01:0.6
    Ks = [get_int(d, 0) for d in ds]
    figure(figsize = (5,3))
    plot(ds, Ks)
```

```
    xlabel("cortical distance (mm)")
    ylabel("K")
    xlim(minimum(ds), maximum(ds))
    ylim(minimum(Ks)-0.05, maximum(Ks)+0.05)
    savefig("figures/plot_K.png", bbox_inches = "tight")
    close()
    return ds, Ks
end

function plot_eigvec(;mu=13)
    #plots the eigenvector corresponding to a given eigenvalue. 13 is
    #the principal eigenvalue
    ds = -0.6:0.01:0.6
    vals = [cos(2*pi*mu* d*512/10 /512) for d in ds]
    figure(figsize = (5,3))
    plot(ds, vals)
    xlabel("cortical distance (mm)")
    ylabel(L"e")
    xlim(minimum(ds), maximum(ds))
    ylim(-1.05, 1.05)
    savefig("figures/plot_eigvec.png", bbox_inches = "tight")
    close()
    return ds, vals
end

function update(W; epsilon = 0.01, K=K, Q=Q)
    #updates the weights a single iteration with learning rate epsilon
    #K: 512x512, W: 512x2, Q: 2x2
    W += epsilon * K * W * Q
    W .+= (1 .- sum(W, dims=2))*0.5 #explicit normalization step
    W[ W.<0 ] .= 0 #threshold at zero
    W[ W.>1 ] .= 1 #threshold at 1
    return W
end

function run_sim(;n = 512, sigma = 0.066, L = 10, N=1000, temp = true,
                 fname="occsimtemp", Plot=true, q_D=0.7)
    #runs simulation of cortical columns for N timesteps.
    K = zeros(n, n)
    for i in 0:(n-1)
        for j in 0:(n-1)
            #construct cortical interaction matrix
            K[i+1, j+1] = get_int( L/(n-1)*i, L/(n-1)*j, sigma=sigma )
        end
    end
    d = Normal(0, 0.01) #gaussian noise
    wLs = 0.5 .+ rand(d, 512) #initialize left weights
    wRs = 1.0 .- wLs #sum of weights is always 1.
    W = [wLs wRs]
    Q = ones(2,2); Q[1,2] = q_D; Q[2,1] = q_D #construct correlation matrix
    stds = zeros(N) #use std as proxy for convergence
    for i = 1:N
        W = update(W, K=K, Q=Q) #update weights
        Wm = W[:,2] -W[:,1] #calculate w_(-)
        stds[i] = std(Wm)
        if temp
            if i%50 == 0 #plot w_(-) every 50 iterations
```

```
                plot_occ(Wm, filename="figures/"*fname*"_int/i"*string(i)*".png")
            end
        end
    end

    Wm = W[:,2] -W[:,1]
    Plot && plot_occ(Wm, filename="figures/"*fname*"_heat.png") #plot result

    figure(figsize=(5,3)) #plot standard deviations
    plot(1:N, stds)
    xlabel("iteration")
    ylabel(L"std({\bf w_-})")
    ylim(0,1)
    savefig("figures/"*fname*"_stds.png", bbox_inches="tight")
    close()

    return W, Wm
end

function plot_mags(mags)
    #plot the magnitudes of the discrete fourier transform of w_(-)
    mus = 0:1:96
    ks = [2*pi/10*mu for mu in mus]
    Ks = [mags[mu+1] for mu in mus]
    figure(figsize = (5,3))
    plot(ks, Ks)
    xlabel("k (1/mm)")
    ylabel("|DFT|")
    xlim(minimum(ks), maximum(ks))
    ylim(0, maximum(Ks*1.05))
    savefig("figures/plot_DFT.png", bbox_inches = "tight")
    close()

end

function repeat_sims(N = 5000)
    #run N simulations, calculate the mean DFT magnitudes and plot this

    dfts = zeros(512,N)
    for i = 1:N
        println("new i: ",i)
        W, Wm = run_sim(Plot=false, temp=false)
        dft = fft(Wm) #perform a fast fourier transform
        mags = norm.(dft) #get magnitude of components
        dfts[:,i] = mags
    end

    mags_mean = mean(dfts, dims=2) #get mean
    plot_mags(mags_mean) #plot result
    writedlm("mags_mean.dlm", mags_mean) #write result to file
    #print optimum value of mu
    println((0:96)[mags_mean[1:97] .== maximum(mags_mean[1:97])], " : ", maximum(mags_mean[1:
    return mags_mean
end

function get_max_k(sig)
    #for a given value of sigma, finds the value of k that maximizes K_tilde
```

```
    ks = 0:0.01:100
    Ks = [get_trans(k, sigma=sig) for k in ks] #get fourier transform at k
    kmax = ks[ Ks .== maximum(Ks) ][1] #find max
    return kmax
end

function scan_widths(sigs = 0.01:0.001:0.200)
    #for a range of sigma values, finds the k value that optimizes
    #K_tilde and plots this result
    N = length(sigs)
    ks = zeros(N)
    for i = 1:N
        k = get_max_k(sigs[i]) # find max k value
        ks[i] = k
    end
    figure(figsize = (5,3)) #plot result
    plot(sigs, ks)
    xlim(minimum(sigs), maximum(sigs))
    ylim(0, maximum(ks))
    plot([0, 1], [8.17, 8.17], "b:")
    ylabel(L"k_{max}")
    xlabel(L"\sigma")
    savefig("figures/test_maxk.png", bbox_inches="tight")
    close()
    return sigs, ks
end

W, Wm = run_sim(fname="occsim_066", sigma=0.066, N=3000) #default parameters
sigs, ks = scan_widths() #find how stripe width varies with k
W, Wm = run_sim(fname="occsim_012", sigma=0.012, N=3000) #narrow stripes
W, Wm = run_sim(fname="occsim_174", sigma=0.174, N=3000) #wide stripes

mags = repeat_sims() #run repeated simulation to get |DFT|

#Plot K, it's fourier transform, and it's principal eigenvector
ds, Ks = plot_Ks()
ks, Ktildes = plot_Ktildes()
ds, vals = plot_eigvec()

#Code for the travelling salesman problem
#Durbin R, Willshaw D (1987)

using PyCall, PyPlot, LinearAlgebra, Distributions,
    Random, DelimitedFiles, LaTeXStrings, StatsBase, Statistics, MultivariateStats
Random.seed!(07031106) #random seed for reproducibility

function swap_cities(cities, N, L0)
    #this function swaps the part of a path between two random cities as
    #required by the simulated annealing algorithm
    cities0 = copy(cities)
    i, j = Rand(1:N, 2) #pick cities to swap
    #get indices of section to reverse
    if j > i inds = [i+1:j-1]
    elseif i > j inds = [i+1:N ; 1:j-1]
    else inds = [1] end

    cities[inds,:] = cities[reverse(inds),:] #reverse path
    L = L0 + #calculate new path length
```

```
        norm(cities[mod(i+1-1, N)+1,:]-cities[i,:]) +
        norm(cities[j,:]-cities[mod(j-1-1)+1,:]) -
        norm(cities0[mod(i+1-1, N)+1,:]-cities0[i,:]) -
        norm(cities0[j,:]-cities0[mod(j-1-1)+1,:])

    return cities, L
end
function sim_annealing(cities0, nmax = 10^7, T0=10)
    #runs a simulated annealing optimzation of the travelling salesman problem
    #nmax is number of iterations, T0 is initial temperature
    N = size(cities)[1]
    cities0 = cities0[shuffle(1:end), :] #random initial path
    L0 = get_length(cities0) #get initial length

    for i=n = 1:nmax
        T = T0/log(n) - T0/log(nmax) #new temperature
        cities, L = swap_cities(cities0, N) #reverse path between two cities
        if L < L0 #always accept if better
            L0, cities0 = L, cities
        else
            p = exp(-(L-L0)/T) #probability of swapping
            if rand()<p
                L0, cities0 = L, cities
            end
        end
    end
    print("final length is: ", L0) #current length
    return cities0, L0
end


function plot_route(cities, points; name="figures/testtpm.png")
    #function for plotting a set of cities and a route through them
    figure()
    points = vcat(points, points[1, :]') #circular route
    plot(cities[:,1], cities[:,2], "ko") #plot cities
    plot(points[:,1], points[:,2], "b-") #plot route
    savefig(name)
    close()
end

function heatmap(results, xs, ys; xlab="", ylab="",
    filename="default", Title="default")
    #given a matrix of z values and lists of x,y values
    #plots a heatmap
    figure()
    imshow(results, cmap=ColorMap("gray_r"), vmin=minimum(results),
            vmax=maximum(results))
    print("plotted")
    xticks(0:(length(xs)-1), xs, rotation=0)
    yticks(0:(length(ys)-1), ys)
    colorbar()
    xlabel(xlab)
    ylabel(ylab)
    title(Title)
    savefig(filename, bbox_inches="tight")
    close()
```

```
end

function update_round(cities, points, K, alpha, beta)
    #performs one update step of the elastic net
    N = size(cities)[1]; M = size(points)[1]
    ws = zeros(N,M)
    for i = 1:N #construct weight matrix
        xi = cities[i,:]
        for j = 2:M
            yj = points[j,:]
            ws[i,j] = exp( -(norm(yj-xi)^2) / (2*K^2) ) #Gaussian attraction
        end
        ws[i,:] /= sum(ws[i,:])
    end

    newpoints = zeros(M,2) #vector for storing new points
    for j = 1:M
        yj = points[j,:] #old position
        delta = alpha * sum( (reshape(ws[:,j], N, 1) .*
                (cities[:,:].- yj')), dims=1 ) #first update term
        #use modulus calculations to reflect circular structure in data
        delta = delta[1,:] + beta*K*(points[mod(j+1-1, M)+1,:]-2*yj +
                points[mod(j-1-1,M)+1,:]) #second update term
        newpoints[j,:] = yj + delta #update point
    end
    return newpoints
end

function get_init_points(cities, M)
    #get a set of M initial points spread evenly around the centroid of N cities
    centroid = mean(cities, dims=1)[1,:]
    points = zeros(M,2) #initialize array
    for i = 1:M
        ang = i/M * 2 * pi #evenly distributed angles
        pos = centroid + (0.099+rand(1)[1]*0.002) .* [cos(ang); sin(ang)] #+noise
        points[i,:] = pos
    end
    return points
end

function get_length(points)
    #get the length of a TSP path specified by points
    points = vcat(points, points[1, :]') #consider circular path
    L = 0
    for i=2:size(points)[1]
        L += norm(points[i,:] - points[i-1,:]) #add length of each line segment
    end
    println("length: ", L)
    return L
end

function TSP(; cities=cities, K = 0.2, alpha=0.2, beta=2.0, Klim=0.001,
             name="test_tsp", decay=0.0005, M=3, Plot=true, plot_int = false)
    #performs an elastic net optimization of the TSP decreasing K exponentially from K
    #with decay rate decay. M specifies the number of points on the path
    #as a multiple of N
    N = size(cities)[1]
```

```
    M = Int(round(M*N)) #number of points on path
    points = get_init_points(cities, M)
    Plot &&& plot_route(cities, points, name="figures/"*name*"_1.png")
    n = 0 #number of iterations
    while K > Klim
        n += 1
        if plot_int
            if n%200 == 0 #plot intermediate routes
                plot_route(cities, points,
                    name="figures/int/"*name*"_K"*string(K)*".png")
            end
        end
        K -= decay*K #exponential decay
        points = update_round(cities, points, K, alpha, beta) #update path
    end
    println(n, " ", K)
    #println(points)
    Plot &&& plot_route(cities, points, name="figures/"*name*"_2.png")
    Plot &&& get_length(points)
    return cities, points
end

function test_parameter_space()
    #investigates the perform of the TSP algorithm as a function of
    #the number of points M on the path and the decay rate lambda
    cities = readdlm("cities.dlm") #read a file of stored cities
    decays = [0.01, 0.005, 0.001, 0.0005, 0.0001]
    Ms = [1.2, 1.5, 2, 3, 5]
    Ls = zeros(length(decays), length(Ms)) #store path lengths
    Ts = zeros(length(decays), length(Ms)) #store completion times
    for (i, decay) in enumerate(decays)
        for (j, M) in enumerate(Ms)
            t = time()
            cities, points = TSP(cities=cities, decay = decay, M = M, name="scan", Plot=false)
            T = time()-t #time for simulation
            Ts[i,j] = T #time
            L = get_length(points) #length
            println("(", decay, " ", M, ")", L, " ", T)
            Ls[i,j] = L
        end
    end
    writedlm("Ls.dlm", Ls) #write data
    writedlm("Ts.dlm", Ts)
    writedlm("Ms.dlm", Ms)
    writedlm("decays.dlm", decays)
    heatmap(Ls, Ms, decays; xlab="M", ylab="lambda", #plot results
        filename="figures/Ls", Title="default")
    heatmap(Ts, Ms, decays; xlab="M", ylab="lambda",
        filename="figures/Ts", Title="default")
    return Ms, decays, Ls, Ts
end

function compare_methods(;n = 30, name = "rand1")
    #compare elastic net to simulated annealing for solving the TSP
    #n specifies number of simulated annealings to run
    if name == "orig"
        cities = readdlm("cities.dlm") #use our pre-defined points
```

```
    else
        cities = rand(100,2) #generate new points and save them for future use
        writedlm("cities_"*name*".dlm", cities)
    end
    t = time()
    cities, points = TSP(name="compare_elastic_"*name, decay=0.0005, M=1.5, plot_int=false)
    Torig = time()-t #time for elastic net (deterministic so only run one simulation)
    Lorig = get_length(points) #length for elastic net

    Ls = zeros(n)
    Ts = zeros(n)
    for i=1:n #try n simulations of simulated annealing
        t = time()
        cities0, L0 = sim_annealing(cities) #run simulated annealing
        Ts[i] = time()-t #store time take
        Ls[i] = L0 #store length
    end
    #print summary data
    println("Lorig: ", Lorig, "    Torig: ", Torig)
    println("Mean: ",mean(Ls),"  std: ",std(Ls),"  min: ",minimum(Ls),"  max: ",maximum(Ls))
    println("Mean: ",mean(Ts),"  std: ",std(Ts),"  min: ",minimum(Ts),"  max: ",maximum(Ts))
end

cities = rand(100,2) #generate cities
#writedlm("cities.dlm", cities) #write to file
cities = readdlm("cities.dlm") #use previously generated cities for ease of comparison

test_parameter_space() #find optimum parameters for elastic net simulation
#run single elastic net simulation
cities, points = TSP(name="test_tsp", decay=0.0005, M=1.5, plot_int=true)
compare_methods() compare elastic net and simulated annelaing

#code for using the elastic net algorithm to model ocular ocular dominance

using PyCall, PyPlot, LinearAlgebra, Distributions,
    Random, DelimitedFiles, LaTeXStrings, StatsBase, Statistics, MultivariateStats

Random.seed!(21031633) #random seed for reproducibility

function plot_pref(cities, points; name = "figures/occ2d")
    #function for plotting results of a simulation
    N = size(cities)[2]
    M = size(points)[1]

    #plot ocular dominance
    zval = points[:, :, 3]
    zval = sign.(zval)
    figure()
    imshow(zval, cmap=ColorMap("gray_r"), vmin = -1,
            vmax = 1)
    xticks([], [])
    yticks([], [])
    savefig(name*"_dominance.png", bbox_inches = "tight")
    close()

    #plot x mapping
    xval = points[:, :, 1]
    figure()
```

```
    imshow(xval, cmap=ColorMap("gray_r"), vmin = minimum(xval),
           vmax = maximum(xval))
    xticks([], [])
    yticks([], [])
    savefig(name*"_xvals.png", bbox_inches = "tight")
    close()

    #plot y mapping
    yval = points[:, :, 2]
    figure()
    imshow(yval, cmap=ColorMap("gray_r"), vmin = minimum(yval),
           vmax = maximum(yval))
    xticks([], [])
    yticks([], [])
    savefig(name*"_yvals.png", bbox_inches = "tight")
    close()

    #plot all points in 3D
    figure()
    plot3D(
        reshape(cities[1,:,:,1], N*N),
        reshape(cities[1,:,:,2], N*N),
        reshape(cities[1,:,:,3], N*N), color=[0,0,1,0.5],
        linestyle = "", marker="o", markersize = 1)
    plot3D(
        reshape(cities[2,:,:,1], N*N),
        reshape(cities[2,:,:,2], N*N),
        reshape(cities[2,:,:,3], N*N), color=[1,0,0,0.5],
        linestyle = "", marker="o", markersize = 1)
    plot3D(
        reshape(points[:,:,1], M*M),
        reshape(points[:,:,2], M*M),
        reshape(points[:,:,3], M*M), color=[1,0,1,0.5],
        linestyle = "", marker="o", markersize = 1)
    savefig(name*"_positions.png", bbox_inches = "tight")
    close()
end

function update_round(cities, points, K, alpha, beta, Print=false)
    #performs one update step of the elastic net
    N = size(cities)[2]; M = size(points)[1]
    ws = zeros(2,N,N,M,M)

    #construct weight matrix
    for k = 1:2 #right/left eye
        for a = 1:N #retinal position
            for b = 1:N
                xab = cities[k,a,b,:]
                for i = 1:M #cortical position
                    for j = 1:M
                        yij = points[i,j,:]
                        ws[k,a,b,i,j] = exp( - (norm(yij-xab)^2) / (2*K^2) )
                        if isnan(ws[k,a,b,i,j])
                            return 0, 0 #things are broken
                        end
                    end
                end
```

```
                Print && println("\n")
                Print && println(ws[k,a,b,:,:])
                ws[k, a, b, :, :] /= sum(ws[k,a,b,:,:]) #normalize
                Print && println(ws[k,a,b,:,:], "    ", sum(ws[k,a,b,:,:]))
            end
        end
    end

    newpoints = zeros(M,M,3) #vector for storing new points
    for i = 1:M
        for j = 1:M
            yij = points[i, j,:] #old position
            delta = zeros(3) #change in position
            for k = 1:2
                for a = 1:N
                    for b = 1:N
                        xab = cities[k,a,b,:] #retinal point
                        delta = delta + alpha * ws[k,a,b,i,j] * (xab - yij)
                    end
                end
            end
            Print && println("\n")
            Print && println(delta)

            #add elastic contributions
            n = 0
            if (j < M) delta = delta + beta*K*points[i, j+1,:]; n+=1 end
            if (i < M) delta = delta + beta*K*points[i+1, j,:]; n+=1 end
            if (j > 1) delta = delta + beta*K*points[i, j-1,:]; n+=1 end
            if (i > 1) delta = delta + beta*K*points[i-1, j,:]; n+=1 end
            delta = delta - n*beta*K*yij

            Print && println(delta)
            newpoints[i, j,:] = yij + delta
        end
    end
    return newpoints
end



function run_sim(cities, cortex, N, M; K = 0.1, alpha=0.2, beta=2.0, Klim=0.001,
            name="occ2d_test", decay=0.001, Plot=true, plot_int = false)
    #performs an elastic net optimization of the TSP decreasing K exponentially from K
    #with decay rate decay. M specifies the number of points on the path
    #as a multiple of N

    Plot && plot_pref(cities, cortex, name="figures/"*name*"_1")
    n = 0 #number of iterations
    while K > Klim
        n += 1
        if n % 10 == 0
            println("n: ", n, "    K: ", K, "    Klim:", Klim)
            if isnan(minimum(cortex)) #simulation diverged
                println("nan encountered, exiting")
                return cities, cortex
            end
```

```
            if n % 100 == 0 #plot intermediate points
                if plot_int
                    plot_pref(cities, cortex, name="figures/int/"*name*"_K"*string(K))
                end
            end
        end
        K -= decay*K #exponential decay
        cortex = update_round(cities, cortex, K, alpha, beta) #update path
    end
    println(n, "  ", K)
    Plot && plot_pref(cities, cortex, name="figures/"*name*"_2")
    return cities, cortex
end

function get_retinae(;N=20, M=40, l =0.075, d = 0.05/2)
    #N^2 is number of points in retinal planes
    #M^2 in cortex plane
    #l is inter-plane separation
    #b is inter-points separation

    #first index specifies retina, next two xy index of point, then coordinates
    cities = zeros(2,N,N,3)
    cortex = zeros(M, M, 3)
    for k = 1:2 #right vs left retina
        for i = 1:N #x location
            for j = 1:N #y location
                cities[k,i,j,:] = [i*d; j*d; l*(3-2*k)]
            end
        end
    end

    #initialize cortical positions uniformly at random in xy
    #uniform distribution in z plane
    for i = 1:M
        for j = 1:M
            cortex[i,j,:] = [rand()*N*d; rand()*N*d; rand()*0.01-0.01/2]
        end
    end

    return cities, cortex
end

#specify parameters (determined empirically)
alpha = 0.2
beta = 2
K = 0.12
N = 20
M = 40
#vary ls for figures
l = 0.15/2
l = 0.2/2
l = 0.1/2
d = 0.05

cities, cortex = get_retinae(N=N, M=M, l = l, d = d) #initialize system
cities, cortex = run_sim(cities, cortex, N, M, K=K, Klim = 0.01,
                    plot_int=true, decay = 0.0005,
```

```
                    name = "N20M40l05d05") #run simulation

##code for the cart pole balancing problem
using PyCall, PyPlot, LinearAlgebra, Distributions,
    Random, DelimitedFiles, LaTeXStrings, StatsBase, Statistics, MultivariateStats

#Physical parameters
#g = -9.8 #m/s2, acceleration due to gravity
g = 9.8
mc = 1.0 #kg, mass of cart
mp = 0.1 #kg, mass of pole,
l = 0.5 #m, half-pole length,
muc = 0.0005#, coefficient of friction of cart on track,
mup = 0.000002#, coefficient of friction of pole on cart,
Ft = 10.0 # p/m 10 newtons, force applied to cart's center of mass at time t.

#model parameters for Barto et al.
alpha = 1000
beta = 0.5
delta = 0.9
gamma = 0.95
lambda = 0.80
sigma = 0.01
dnorm = Normal(0, sigma^2)
dt = 0.02

function heatmap(results, xs, ys; xlab="", ylab="",
    filename="default", Title="default")
    #given a matrix of z values and lists of x,y values
    #plots a heatmap
    figure()
    imshow(results, cmap=ColorMap("gray_r"), vmin = minimum(results),
            vmax = maximum(results))
    print("plotted")
    xticks(0:(length(xs)-1), xs, rotation=0)
    yticks(0:(length(ys)-1), ys)
    colorbar()
    xlabel(xlab)
    ylabel(ylab)
    title(Title)
    savefig(filename, bbox_inches = "tight")
    close()
end


function plot_poles(N, xs, thetas, performance, dt, fname, vs; all=true)
    #N is number of timesteps, xs and thetas are given for a simulation
    #performance is simulation length by trial number

    if all #make all plots
    #plot cart position vs time
    ts = (1:N)*dt
    figure(figsize = (5,3))
    plot(ts, xs)
    plot(ts, repeat([-2.4], N), "k—")
    plot(ts, repeat([2.4], N), "k—")
    xlabel("time (s)")
    ylabel("x")
```

```
    savefig("figures/"*fname*"_xs.png", bbox_inches = "tight")
    close()

    #plot theta vs time
    figure(figsize = (5,3))
    plot(ts, thetas)
    plot(ts, repeat([-12], N), "k--")
    plot(ts, repeat([12], N), "k--")
    xlabel("time (s)")
    ylabel("theta")
    savefig("figures/"*fname*"_thetas.png", bbox_inches = "tight")
    close()

    #plot both x and theta
    figure(figsize = (5,3))
    plot(ts, xs*(1/2.4), "b:")
    plot(ts, thetas*(1/12), "r:")
    plot(ts, repeat([-1], N), "k--")
    plot(ts, repeat([1], N), "k--")
    legend(["cart position", "pole angle"])
    xlabel("time (s)")
    ylabel("theta")
    savefig("figures/"*fname*"_xs_thetas.png", bbox_inches = "tight")
    close()

    if length(vs) > 5
        #plot heatmap of predicted reward
        newvs = zeros(3, 6)
        for (i,thetadot) = enumerate([-100;0;100])
            for (j,theta) = enumerate([-9; -3.5; -0.5; 0.5; 3.5; 9])
                temps = zeros()
                for x = [-1.6;0;1.6] #average over x and xdot
                    for xdot = [-1;0;1]
                        temps = [temps; vs[get_state([x;theta;xdot;thetadot])]]
                    end
                end
                newvs[i,j] = mean(temps)
            end
        end
        heatmap(newvs, [-9; -3.5; -0.5; 0.5; 3.5; 9], [-100;0;100];
                xlab=L"$\theta$", ylab=L"$\dot \theta$",
                filename="figures/"*fname*"_vs.png", Title="Predicted Reward")
    end

    end

    #plot performance vs trial number
    ntrial = length(performance)
    figure(figsize = (5,3))
    plot(1:ntrial, performance)
    xlabel("trial")
    ylabel("time (s)")
    savefig("figures/"*fname*"_performance.png", bbox_inches = "tight")
    close()
end

function smooth_performance(performance, dt)
```

```
    #compute rolling average of trial length as in Barto et al.
    N = length(performance)
    newperf = zeros(N)
    for i = 1:4
        newperf[i] = mean(performance[1:i])
    end
    for i = 5:N #average over 5 timepoints
        newperf[i] = mean(performance[i-4:i])
    end
    return newperf*dt #real time
end

function get_state(state)
    #return unique state index given vector of [x,theta,xdot,thetadot]
    x, theta, xdot, thetadot = state

    k1 = sum(x .> [-0.8, 0.8])
    k2 = sum(theta .> [-6,-1,0,1,6])
    k3 = sum(xdot .> [-0.5, 0.5])
    k4 = sum(thetadot .> [-50, 50])
    k = 54*k1 + 9*k2 + 3*k3 + k4 + 1
    return k #index of non-zero element
end

function get_output(state, ws; d=dnorm)
    #get control action. +1 is right, -1 is left
    k = get_state(state)
    y = sign(ws[k] + rand(d)) #get output
    return y, k
end

function update_es(es, k, y, delta)
    #Update ASE eligibility trace. k is index of current state
    es *= delta #decay
    es[k] += (1-delta)*y #eligibility trace increased for current state
    return es
end

function update_xbars(xbars, k, lambda)
    #Update ACE eligibility trace. k is index of current state.
    xbars *= lambda #decay
    xbars[k] += (1-lambda) #eligibility trace increased for current state
    return xbars
end

function update_ws(ws, r, es, alpha)
    #update ASE weights
    ws += alpha*r*es
    return ws
end

function update_vs(vs, r, xbars, beta)
    #update ACE weights
    vs += beta*( r )*xbars
    return vs
end
```

```
function update_rhat(r, pt, ptm, gamma)
    #pt = p(t). ptm=p(t-1)
    rhatt = r + gamma*pt-ptm
    return rhatt
end

function update_cart(state, yt; dt = 0.02, l=l, mc=mc, mp=mp, muc=muc, mup=mup, g=g)
    #update the physical system according to equations in Barto et al.'s appendix
    x, theta, xdot, thetadot = state
    theta, thetadot = 2*pi/360*theta, 2*pi/360*thetadot #convert to radians
    Ft = 10*yt #force to apply

    dthetadot = ( g*sin(theta) +
                cos(theta)*( ( -Ft-mp*l*thetadot^2*sin(theta) + muc*sign(xdot) )/( mc + mp ) )
                mup*thetadot/(mp*l) ) / (
                l*( 4/3 - ( mp* (cos(theta)^2) )/( mc+mp ) ))
    dxdot = (  Ft + mp*l*(thetadot^2*sin(theta) - dthetadot*cos(theta)) -
            muc*sign(xdot) ) / ( mc+mp )

    #update state
    thetadot += dthetadot*dt
    xdot += dxdot*dt
    theta += thetadot*dt
    x += xdot*dt
    return [x; 360/(2*pi)*theta; xdot; 360/(2*pi)*thetadot]
end

function state_failed(state)
    #find out if a given state is outside the system boundaries
    if abs(state[1]) > 2.4 || abs(state[2]) > 12
        return true #system is in a failure state
    end
    return false #state is allowed

end

function update_2layer(state, ws, vs, fs, D, cs, A, ptm, oldfailed; d=dnorm,
                        alpha=alpha, lambda=lambda, gamma=gamma, delta=delta, beta=beta,
                        l=l, mc=mc, mp=mp, muc=muc, mup=mup, layer1=false)
    #perform a single iteration of the 2-layer system from Anderson 1987
    #if 1layer = true, use only a single layer of this system

    #get input activities. x5 is constant bias term
    xs = [(state[1]+2.4)/4.8;(state[2]+12)/24;(state[3]+1.5)/3;(state[4]+115)/230;0.5]
    zs = 1 ./ (1 .+ exp.(-D*xs)) #ASE hidden layer

    P = ws' * xs + fs' * zs
    P = 1 / (1 + exp(-P)) #probability of rightwards impulse
    if rand() < P
        yt = 1
    else
        yt = -1
    end

    qstt = 1 ./ (1 .+ exp.(-A*xs)) #ACE hidden layer
    ptt = vs' * xs + cs' * qstt #predicted reward
```

```
    state = update_cart(state, yt, l=l, mc=mc, mp=mp, muc=muc, mup=mup) #state t+1
    #get new input values
    newxs = [(state[1]+2.4)/4.8;(state[2]+12)/24;(state[3]+1.5)/3;(state[4]+115)/230;0.5]

    qstp = 1 ./ (1 .+ exp.(-A*newxs)) #q[t,t+1]
    ptp = vs' * newxs + cs' * qstp #p[t,t+1]

    #calculated ACE reward signal
    if oldfailed #we failed on our previous iteration
        rhatt = 0
        failed = false
    elseif state_failed(state) #current state is a failed state
        rhatt = -1 - ptt
        failed = true
    else
        rhatt = gamma*ptp - ptt
        failed = false
    end
    qs = qstt
    pt = ptt

    #update ASE weights
    ws += alpha*rhatt*(max(yt,0) - P)*xs
    D += ( ((0.2*alpha) * rhatt * zs) .* (1 .- zs) .* sign.(fs) * (max(yt,0) - P) ) * xs'
    fs += alpha*rhatt*(max(yt,0) - P)*zs

    #update ACE weights
    vs += beta*rhatt*xs
    A += ( ((0.25*beta)*rhatt*qs) .* (1 .- qs) .* sign.(cs) ) * xs'
    cs += beta * rhatt * qs

    if failed #initialize new state
        state = [rand()*4.8-2.4;rand()*24-12;0;0]
    end
    if layer1 #set second layer weights to 0
        D = zeros(5,5); fs = zeros(5); A = zeros(5,5); cs = zeros(5)
    end

    return state, ws, vs, fs, D, cs, A, pt, yt, P, zs, qs, xs, failed
end

function update_system(state, ws, vs, es, xbars, ptm; d=dnorm,
                       alpha=alpha, lambda=lambda, gamma=gamma, delta=delta, beta=beta,
                       l=l, mc=mc, mp=mp, muc=muc, mup=mup)
    #update the system from Barto et al.

    yt, k = get_output(state, ws) #get impulse at time t
    pt = vs[k] #get probability of future reward at time t
    if state == [0;0;0;0] rhatt = 0 else
    rhatt = gamma*pt - ptm #calculate expected reward at time t
    end

    ws = update_ws(ws, rhatt, es, alpha) #update ASE weights t+1
    vs = update_vs(vs, rhatt, xbars, beta) #update ACE weights t+1
    es = update_es(es, k, yt, delta) #update ASE eligibility traces t+1
    xbars = update_xbars(xbars, k, lambda) #update ACE eligibility traces t+1
```

```
    state = update_cart(state, yt, l=l, mc=mc, mp=mp, muc=muc, mup=mup) #update states to t+1

    return state, ws, vs, es, xbars, pt, k
end

function run_sim(;fname="test", N = 500000, ntrials = 100, mc=mc, mp=mp, l=l, muc=muc,
                 mup=mup, layer2 = false, alpha=alpha, beta=beta, gamma=gamma,
                 lambda=lambda, delta=delta, d=dnorm, Plot=true, Print=true,
                 all = false, layer1 = false)
    #run a simulation for N timesteps of ntrials trials, whichever occurs first
    #layer2=true for an Anderson simulation, layer1=true to do this with 1 layer

    n = 162 #number of Barto states
    pt = 0 #no initial predicted reward

    if layer2 #Anderson
        A, D = zeros(5,5), zeros(5,5)
        ws, vs, fs, cs = zeros(5), zeros(5), zeros(5), zeros(5)
    else
        ws, vs, es, xbars = zeros(n), zeros(n), zeros(n), zeros(n)
    end
    state = [0;0;0;0] #start upright
    ntrial, lasttrial, performance = 0, 0, zeros(ntrials) #keep track of performance
    thetas = zeros(N) #store angles
    xs = zeros(N) #store positions
    Print && println("\nnew simulation")
    failed = true #start from a newly initialized state; history irrelevant
    for i = 1:N
        if !layer2 #Barto update step
        state, ws, vs, es, xbars, pt, k = update_system(state, ws, vs, es, xbars, pt,
                                                        l=l, mc=mc, mp=mp, muc=muc, mup=mup,
                                                        alpha=alpha, beta=beta, gamma=gamma,
                                                        lambda=lambda, delta=delta)
        else #Anderson update step
        state, ws, vs, fs, D, cs, A, pt, yt, P, zs, qs, xvals, failed = update_2layer(state, w
                                                             l=l, mc=mc, mp=mp, muc=muc, mup=mup,
                                                             alpha=alpha, beta=beta, gamma=gamma,
                                                             lambda=lambda, delta=delta, layer1 = l
        end
        xs[i] = state[1] #store data
        thetas[i] = state[2]
        if failed #if failed Anderson state, update trial number
            ntrial += 1
            Print && println("system failed i=",i,"  ntrial=",ntrial)
            performance[ntrial] = i-lasttrial
            lasttrial = i
            if ntrial == ntrials break end
        end
        if (abs(state[1]) >= 2.4 || abs(state[2]) >= 12) && !layer2 #failed Barto
            ntrial += 1
            #print what happened
            if state[1] <= -2.4
                Print && println("cart hit left, i=",i,"  ntrial=",ntrial)
            elseif state[1] >= 2.4
                Print && println("cart hit right, i=",i,"  ntrial=",ntrial)
            elseif state[2] <= -12
                Print && println("pole fell left, i=",i,"  ntrial=",ntrial)
```

```
            else
                Print && println("pole fell right, i=",i,"  ntrial=",ntrial)
            end

            #we're now not in any of the 162 states
            rhatt = -1-pt
            pt = 0
            #learn from error signal
            ws = update_ws(ws, rhatt, es, alpha) #update ASE weights t+1
            vs = update_vs(vs, rhatt, xbars, beta) #update ACE weights t+1
            es = zeros(n) #update_es(es, k, yt, delta) #update ASE eligibility traces t+1
            xbars = update_xbars(xbars, k, lambda)

            state = [0;0;0;0] #-1 #reset system
            performance[ntrial] = i-lasttrial #store performance of last trial
            lasttrial = i
            if ntrial == ntrials break end
        end
    end

    if ntrial < ntrials #if we reached iteration threshold, extrapolate performance
        performance[ntrial+1:end] .= max(performance[ntrial], N-lasttrial)
    end
    performance = smooth_performance(performance, dt) #smooth performance as in Barto
    Plot && plot_poles(N, xs, thetas, performance, dt, fname, vs; all=all)

    return performance
end

function test_performance(;n=10, ntrials=50, mc=mc, mp=mp, l=l, muc=muc, mup=mup,
                fname="test", Plot=true, alpha=alpha, beta=beta, gamma=gamma,
                lambda=lambda, delta=delta, d=dnorm)
    #run n simulations with maximum ntrial trials and average performance
    perfs = zeros(ntrials, n) #performances
    learns = zeros(n) #trials needed to learn
    for i = 1:n
        perf = run_sim(;ntrials = ntrials, mc=mc, mp=mp, l=l, muc=muc, mup=mup, Plot=false,
                alpha=alpha, beta=beta, gamma=gamma, lambda=lambda, delta=delta, Print=true)
                #run simulation
        perfs[:,i] = perf
        learn = sum(perf .< 500000/10*0.02) #define learning as threshold performance
        learns[i] = learn
    end

    avgperf = mean(perfs, dims=2)
    if Plot #plot average performance
        figure(figsize = (5,3))
        plot(1:ntrials, avgperf)
        xlabel("trial")
        ylabel("time (s)")
        savefig("figures/"*fname*"_meanperformance.png", bbox_inches = "tight")
        close()
    end

    learn = mean(learns) #get mean learning rate
    println("learned after ", learn, " trials")
    return learn
```

```
end

function test_params(params1, params2, name1, name2; fname="test", n=100, ntrials=200)
    #test the effect of parameters on the performance of the system
    #paramsi, name1 specify the set of parameters used and which parameter to vary
    N, M = length(params1), length(params2)
    learns = zeros(N,M)
    for (i,p1) in enumerate(params1)
        for (j,p2) in enumerate(params2) #consider every parameter combination
            if (name1, name2) == ("delta", "lambda")
                learn = test_performance(n=n, ntrials=ntrials, Plot=false,
                                delta=p1, lambda=p2)
            elseif (name1, name2) == ("alpha", "beta")
                learn = test_performance(n=n, ntrials=ntrials, Plot=false,
                                alpha=p1, beta=p2)
            elseif (name1, name2) == ("mp", "l")
                learn = test_performance(n=n, ntrials=ntrials, Plot=false,
                                        mp=p1, l=p2)
            end
            learns[i,j] = learn #store performance
            print(name1,": ", p1, "  ",name2,": ",p2)
        end
    end
    heatmap(learns, params2, params1; xlab=name2, ylab=name1,
        filename="figures/"*fname*"_heat.png", Title=name1*" "*name2) #plot
    writedlm(fname*"_param1.dlm", params1)
    writedlm(fname*"_param2.dlm", params2)
    writedlm(fname*"_results.dlm", learns)
end

function compare_1_2_layer(N=500000, n = 100, ntrials = 5000, fname="compare_layers")
    #compare the performance of one- and two-layer Anderson networks
    perfs1 = zeros(ntrials, n)
    perfs2 = zeros(ntrials, n)
    for i = 1:n #average performance of n trials
        println("new i:", i)
        perf1 = run_sim(;ntrials = ntrials, mc=mc, mp=mp, l=l, muc=muc, mup=mup,
                Plot=false, alpha=1, beta=0.2, gamma=0.9, Print=false,
                layer2 = true, layer1=true) #run 1-layer simulation
        perfs1[:,i] = perf1
        perf2 = run_sim(;ntrials = ntrials, mc=mc, mp=mp, l=l, muc=muc, mup=mup,
                Plot=false, alpha=1, beta=0.2, gamma=0.9, Print=false,
                layer2 = true, layer1=false) #run 2-layer simulation
        perfs2[:,i] = perf2
    end
    #average performance
    avgperf1 = mean(perfs1, dims=2)
    avgperf2 = mean(perfs2, dims=2)

    #plot result
    figure(figsize = (6,2))
    semilogy()
    plot(1:ntrials, avgperf1, "r-")
    plot(1:ntrials, avgperf2, "b--")
    legend(["1 layer", "2 layers"])
    xlabel("trial")
    ylabel("time (s)")
```

```
    savefig("figures/"*fname*".png", bbox_inches = "tight")
    close()
end

run_sim(fname="test", N=500000, ntrials=100, all=true) #run Barto simulation
test_performance(n=100, fname="real") #average over 100 simulations

test_params([0.4;0.6;0.8;0.9;0.99],[0.4;0.6;0.8;0.9;0.99],"delta", "lambda",
            fname="test_delta_lambda_0") #test the effect of delta and lambda
test_params([0.001;1; 10; 1000; 10000],[0.05;0.25;0.5;0.75;1.0],"alpha", "beta",
            fname="test_alpha_beta") #alpha and beta
test_params([0.01;0.03;0.1;0.3;1],[0.05;0.2;0.5;2;10],"mp", "l", fname="test_mp_l")

compare_1_2_layer() #compare Anderson networks with 1 and 2 layers
```