Computational Neruscience 1

USN: 303039534

1 Hodgkin Huxley model

We implement the Hodgkin Huxley model as described in Dayan and Abbott (2005), defined by equations 1-5

$$c_m \frac{dV}{dt} = -i_m + \frac{I_e}{A} \tag{1}$$

$$i_m = \bar{g}_L(V - E_L) + \bar{g}_K n^4 (V - E_K) + \bar{g}_{Na} m^3 h (V - E_{Na}) \tag{2}$$

$$\tau_n(V) = \frac{1}{\alpha_n(V) + \beta_n(V)} \tag{3}$$

$$n_{\text{inf}} = \alpha_n(V)\tau_n(V) \tag{4}$$

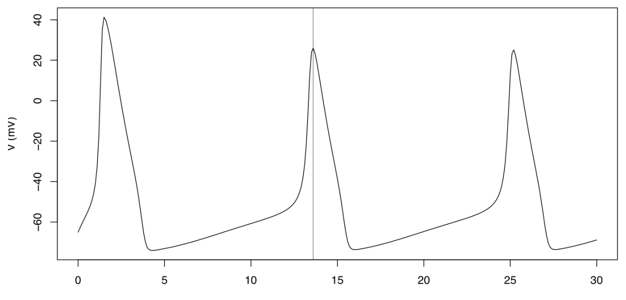
$$\tau_n(V)\frac{dn}{dt} = n_{\text{inf}}(V) - n \tag{5}$$

With $\alpha_n(V)$ and $\beta_n(V)$ given in Dayan and Abbott, and including equivalent equations to (3)-(5) for gating variables m and h. Empirical parameters and their values taken from either the assignment description or Dayan and Abbott are given in table 1

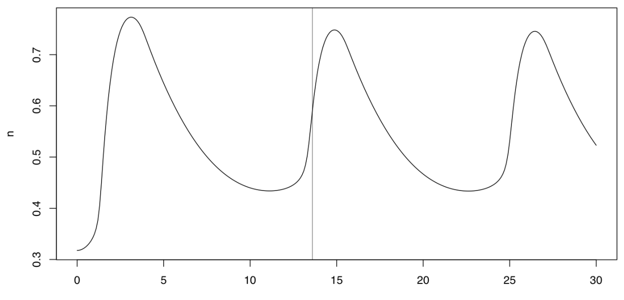
c_m	$\bar{g}_L$	$\bar{g}_K$	$\bar{g}_{Na}$	E_L	E_K	E_{Na}	V_0	n_0	m_0	h_0
$10\frac{nF}{mm^2}$	$3\frac{\mu S}{mm^2}$	$360\frac{\mu S}{mm^2}$	$1200\frac{\mu S}{mm^2}$	$-54.387mV$	$-77mV$	$50mV$	$-65mV$	0.3177	0.0529	0.5961

Table 1: Fixed parameters for Hodgkin-Huxley model

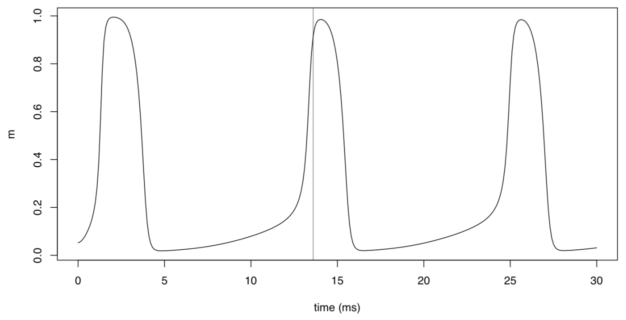
The external input I_e/A is varied over the course of the assignment. We integrate equations (1) and (5) using the R *deSolve* library with a timestep of $dt = 0.1$ as suggested in the assignment. Setting $I_e/A = 200nA/mm^2$, we get spiking activity in our model neurons with a firing rate of approximately 80 hz. The model potential and gating variables are plotted for a 30ms simulation in figure 1.



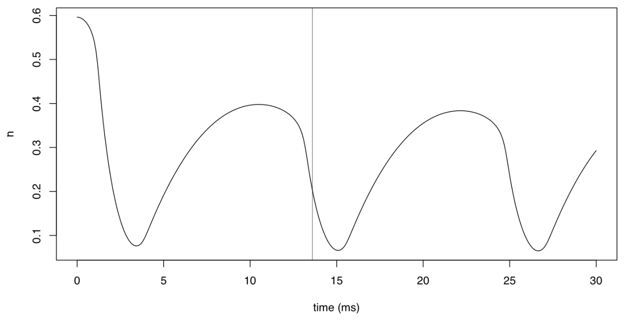
(a) Voltage over time for a Hodking Huxley simulation



(b) Potassium gating variable n



(c) Sodium gating variable m



(d) Sodium inactivation variable h

We see that as the potential increases, so does the sodium gating variable m, leading to increased sodium influx and an increase in the potential towards the sodium equilibrium potential of $E_{Na} = 50mV$. However, this also drives the sodium inactivation variable h to 0 with a longer time constant, which terminates the sodium influx. In combination with an increase in the value of the potassium gating variable m, this leads to a decrease in the potential as it approaches the potassium equilibrium potential of $E_K = -77mV$.

In figure 1 the apiking activity is driven by the external input I_E/A, and we can investigate the effect of this variable on the behavior of the system (**??**). If we denote the time difference between the two final spikes of our system as ΔT, we define the equilibrium firing rate as $\omega = \frac{1}{\Delta T}$ and require the final spike to occur within ΔT of terminating the simulation to ensure sustained firing. In this case we do not need to average over multiple spikes since the simulation is deterministic with no stochastic component. We define a spike as a local maximum in V with V¿0.

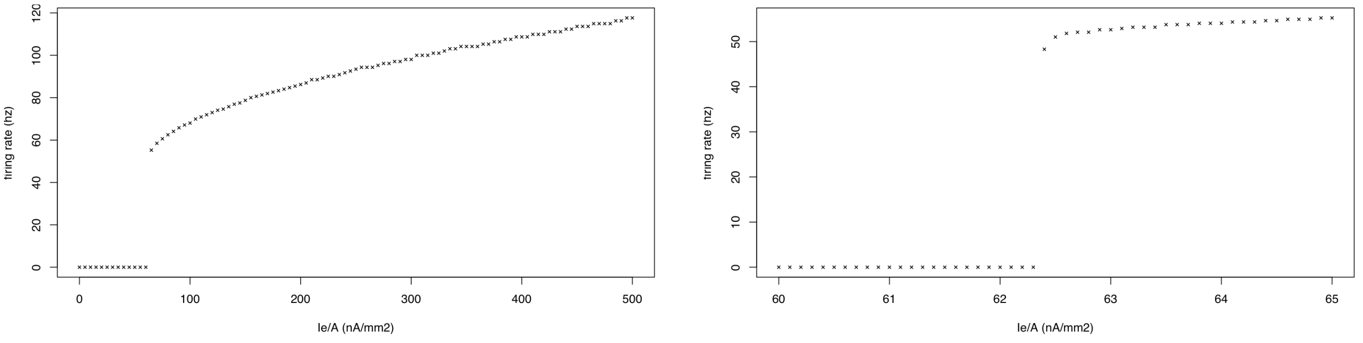


Figure 2: Sustained firing rate as a function of external input. *left:* scanning the range $I_e/A \in [0 : 500]$, we find a firing rate of zero at low input and a firing rate linear in I_e/A at high input. *right:* zooming in on the region $I_e/A \in [60 : 65]$ we find a discontinous jump in the firing rate from 0 to 50 between 62.3 and $62.4nA/mm^2$.

We see from figure **??**a that for high external inputs, the firing rate is approximately linear in I_e/A. For small external input, the leak from the neuron is sufficient to prevent the potential from reaching threshold and we thus see a firing rate of 0. We find that there is a switch between these two behaviors between $I_e/A = 60$ and $I_e/A = 65$ and expand this region in figure **??**b. Even at this high resolution, we find the transition to be discontinuous; if $I_e/A < 62.35$ there is no firing and the neuron is at steady state, while if $I_e/A > 62.35$ the system exceeds threshold and we get a finite firing rate of 50 hz.

Finally we consider what happens in the case of an inhibitory external input with $I_e/A = -50nA/mm^2$ for 5 ms followed by removing all external input. The current and gating variables for such a system are plotted in figure 3.

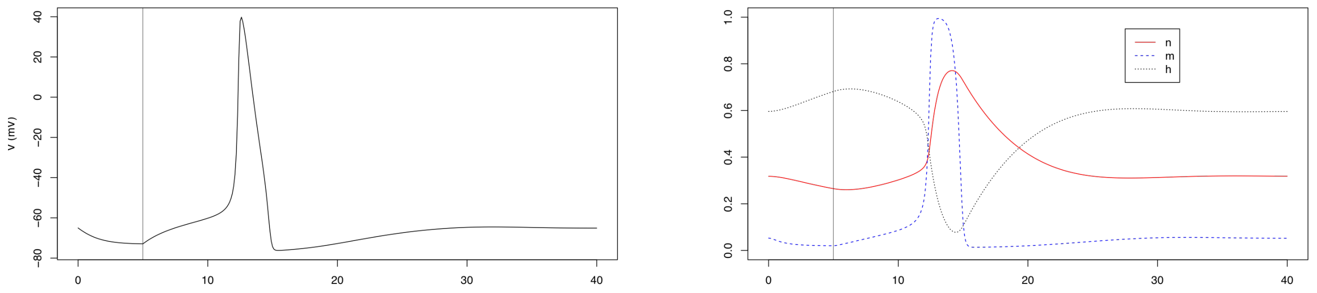


Figure 3: Hodgkin-Huxley model showing a post-inhibitory rebound following brief repression. (*left*) potential as a function of time. (*right*) gating variables n, m & h. Dashed line indicates $t = 5ms$ when repression is relieved.

Surprisingly, we find that the neuron fires exactly once after the removal of the external input, following which it returns to steady state. We can understand this behavoir by looking at what happens to the gating variables. At $t = 5ms$ following inhibition, m has been driven to near 0 while $h = 0.68$. This mean that upon relief of inhibition, conditions are such that there is a rapid influx of sodium at a shorter timescale than $\tau_n = 5.75$ and $\tau_h = 7.93$ under these conditions. The neuron therefore reaches threshold and fires a spike. However, after the refractory period there is no input to drive the neuron to threshold again, and it reaches an equilibrium state of $-65mV$ where $i_m = 0$.

2 Coupled integrate and fire neurons

We implement a set of two coupled integrate and fire neurons. These evolve over time according to

$$\tau_m \frac{dV}{dt} = E_L - V - r_m \bar{g}_s P_s (V - E_S) + R_m I_e \tag{6}$$

Here, r_m is the membrane resistance, $\bar{g}_s$ is the synaptic conductance when the synapse is active, P_s is the probability of the synapse being open, and $R_m I_e$ is the external input.

We further model P_s according to equations

$$\tau_s \frac{dP}{dt} = \exp[1] P_{max} z - Ps \tag{7}$$

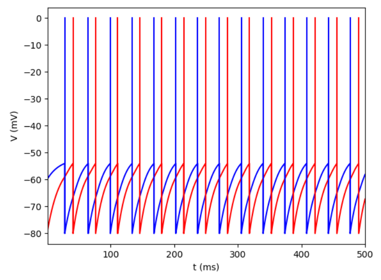
$$\tau_s \frac{dz}{dt} = -z \tag{8}$$

Here z_i is a synaptic gating variable that is set to 1 whenever neuron j fires. We set dt to 0.005 since decreasing dt further does not affect spike times in the case of either excitatory or inhibitory connections. We further use parameters in table 2 unless otherwise stated.

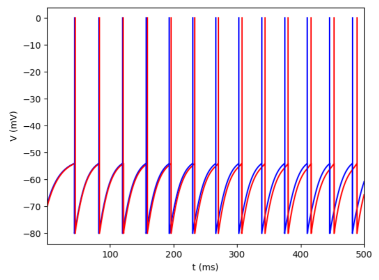
E_L	V_{th}	V_{reset}	τ_m	$r_m\bar{g}_s$	$R_m I_e$	τ_s	P_{max}	$E_{S;excitatory}$	$E_{S,inhibitory}$
$-70mV$	$-54mV$	$-80mV$	$20ms$	0.15	$18mV$	$10ms$	0.5	$0mV$	$-80mV$

Table 2: Parameters for integrate and fire model

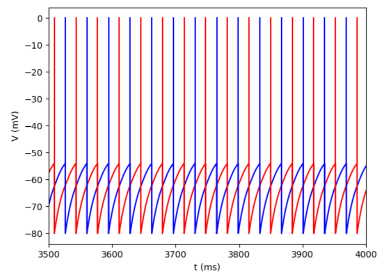
To investigate this system in a simple setting, we connect the two neurons via excitatory synapses and plot the voltage of each neuron as a function of time (figure 4).



(a) Voltage of two excitatory for the first 500 ms of a simulation with $V_{init} = (-60, -80)$



(b) Voltage of two excitatory for the first 500 ms of a simulation with $V_{init} = (-71, -72)$



(c) Voltage of two excitatory for 500 ms after 3500 ms of a simulation with $V_{init} = (-71, -72)$

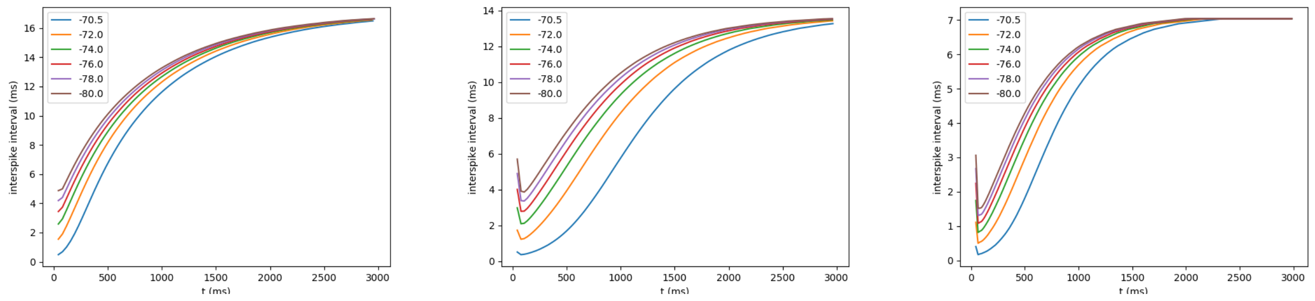
Figure 4: Excitatory neurons

We see that in the case of excitatory neurons with default parameters, it appears that large differences in initial conditions initially lead to alternate firing while very similar initial conditions lead to near-synchronous firing. However, if we let the simulation run until threshold is reached at $t = 3500$ when interspike intervals become constant, we find that the simulation with $V_{init} = (-71, -72)$ also exhibits alternate firing. It thus appears that the equilibrium state of the system depends on the hyperparameters in table 2 but not on the initial conditions.

To investigate the effects of initial conditions, the synaptic time constant τ_s and the synaptic strength ḡs further, we quantify the interspike interval as a function of time for different conditions. Here, the interspike interval is defined as the phase difference between neuron 1 and neuron 2 and quantified by looking at the difference in time to the nearest spike of neuron 2 for a given spike in neuron 1.

We see in figure 5 that for the default parameters, the initial potentials affect the interspike interval at early times, but that this difference eventually disappears since all the simulations approach an interspike interval of 16 ms corresponding to alternate firing.

The same pattern is observed when increasing τ_s from 10ms to 20ms where differences in interspike interval also decrease over time. However, in this case the interspike intervals converge more slowly as would be expected from slower coupling between the neurons. We find that they also converge to a lower interspike interval due to a higher equilibrium firing rate. This is at first surprising given the increased time constant, but on a second thought reasonable as the larger time constant broadens the synaptic current curve and leads to higher total current flow between the two neurons over the course of a synaptic event.

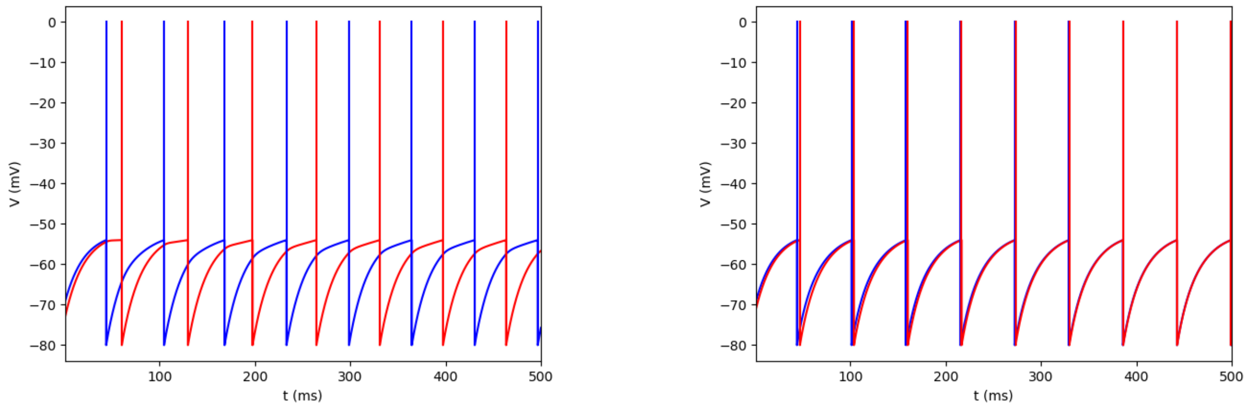


(a) $\tau_s = 10$ (b) $\tau_s = 20$ (c) $\tau_s = 10\ rmgs = 0.5$

Figure 5: Interspike interval over time for a pair of excitatory neurons with different physical parameters. In every case, neuron 1 was initialized at $-70mV$ and each line corresponds to a different V_init of neuron 2, ranging from -70.5 to -80.

When increasing the synaptic strength from figure 5a to figure 5c we also observe an increased firing rate as is expected from increasing the current flow between two excitatory neurons. The neurons with a higher synaptic strength also converge to alternate firing quicker than in figure 5a .

We can perform similar analyses for inhibitory neurons, with equivalent plots to figures 4 and 5 given in figures 6 and **??**. In contrast to the excitatory neurons, we find the long-term behavior of a system of inhibitory neurons to depend strongly on the initial state of the system.

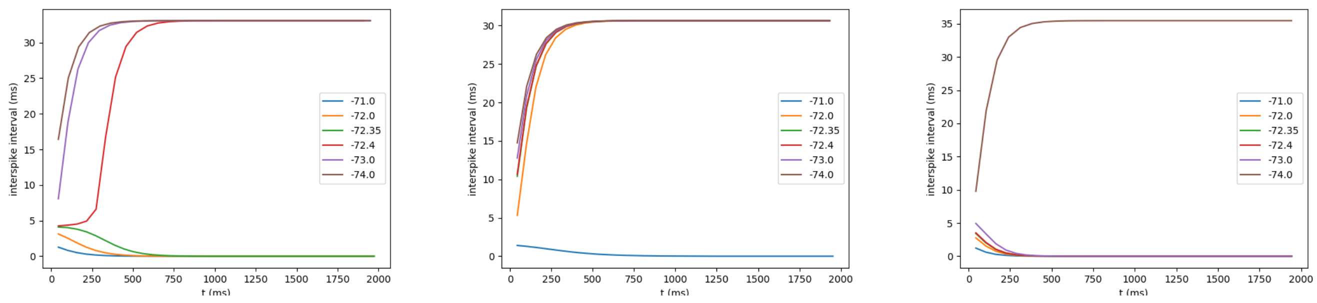


(a) Voltage over time for Vinit = -70, -74 (b) Voltage over time for Vinit = -70, -72

Figure 6: Inhibitory neurons

We see that for $\tau_s = 10ms$ and $V_{init}(1) = -70mV$, a difference in initial potentials between the two neurons of $-2.35mV$ leads to long-term synchronous firing while a potential difference larger than $-2.40mV$ leads to alternate firing. We also note that convergence in this case is much quicker than for the excitatory neurons. How similar the initial potentials must be for synchronous firing depends strongly on τ_s, with shorter time constants requiring more similar initial potentials (figure 7b) and longer time constants exhibiting synchronous firing for more dissimilar potentials (figure 7c).

Of course we cannot qualitatively investigate plots as the above for every possible set of parameters and initial conditions. Instead, we let $\tau_s \in [5, 10, 15]$ and $r_m\bar{g}_s \in [0.05, 0.15, 0.30]$ for both excitatory and inhibitory neurons and



(a) $\tau_s = 10$ (b) $\tau_s = 7$ (c) $\tau_s = 13$

Figure 7: Interspike interval over time for a pair of inhibitory neurons given different τ_s values. In every case, neuron 1 was initialized at $-70mV$ and each line corresponds to a different V_init of neuron 2, ranging from -71 to $-74mV$. If a line goes to zero, it indicates that the long term behavior of the system is synchronous firing.

run 75 simulations for each parameter set with V_1 and V_2 initialized uniformly at random in $[-55, -80]$. We then quantify the firing rate and interspike interval at equilibrium for each simulation. In this case, we run each simulation for $10000ms$ rather than assessing convergence during the simulation, since this is sufficiently long to ensure convergence for all parameter sets as evident from figure **??**.

Similarly to the Hodgkin Huxley model, if the time difference between the final two spikes of neuron 1 is ΔT, we calculate the firing rate as $\omega = \dfrac{1}{\Delta T}$ and require the final spike to occur within ΔT of terminating the simulation. In this case we also do not need to average over multiple spikes since the simulation is deterministic with no stochastic component.

Scatterplots of the equilibrium quantities for all 75 simulations at each parameter set are given in figure **??**. The parameter set $(E_s = -80, \tau_s = 0.15, r_m\bar{g}_s = 0.30)$ is not included since in the case of strong inhibitory synapses with large time constants, the firing of neuron 2 is completely supressed by neuron 1, and only neuron 1 fires.

We see from figure **??** that the long term behavior of an excitatory system is completely determined by the physical parameters and is independent of initial conditions. We find that the equilibrium firing rate increases with both τ_s and $r_m\bar{g}_s$ as discussed above, and that the firing of our two neurons becomes increasingly less synchronous with τ_s and $r_m\bar{g}_s$. Note that while the interspike interval decreases e.g. from $(0, 10, 0.05)$ to $(0, 10, 0.15)$, this still corresponds to more synchronous firing due to the increase in firing rate. For the $(0, 5, 0.05)$ system, firing is almost synchronous, with neuron 1 driving firing of neuron 2 shortly afterwards.

However, this near-synchronous behavior is fundamentally different from the behavior of inhibitory neurons where we can observe firing patterns that are *exactly* synchronous in the long time limit. We see that the proportion of synchronous firing increases with τ_s but decreases with $r_m\bar{g}_s$ as discussed above. However, while the frequency of synchronous firing decreases with $r_m\bar{g}_s$, the difference in firing rate between alternate firing and synchronous firing increases with $r_m\bar{g}_s$ since the stronger synapses lead to stronger inhibition between the alternately firing neurons and thus a relatively lower firing rate.

3 Networks

It is common to separate the neurons of a network model into a population of excitatory neurons and a population of inhibitory neurons, allowing us to describe the time evolution of the network by two differential equations

$$\tau_E \frac{dv_E}{dt} = -v_E + [M_{EE}v_E + M_{EI}v_I - \gamma_E]_+ \tag{9}$$

$$\tau_I \frac{dv_I}{dt} = -v_I + [M_{IE}v_E + M_{II}v_I - \gamma_I]_+ \tag{10}$$

Here, γ is an external input which could be e.g. a previous layer of neurons. M_{AB} specifies the input from population B to population A, and given the division into inhibitory and excitory neurons, we require $M_{AI} \leq 0$ and $M_{AE} \geq 0$. In the present case we describe the excitatory and inhibitory populations by a single firing rate each, but we could

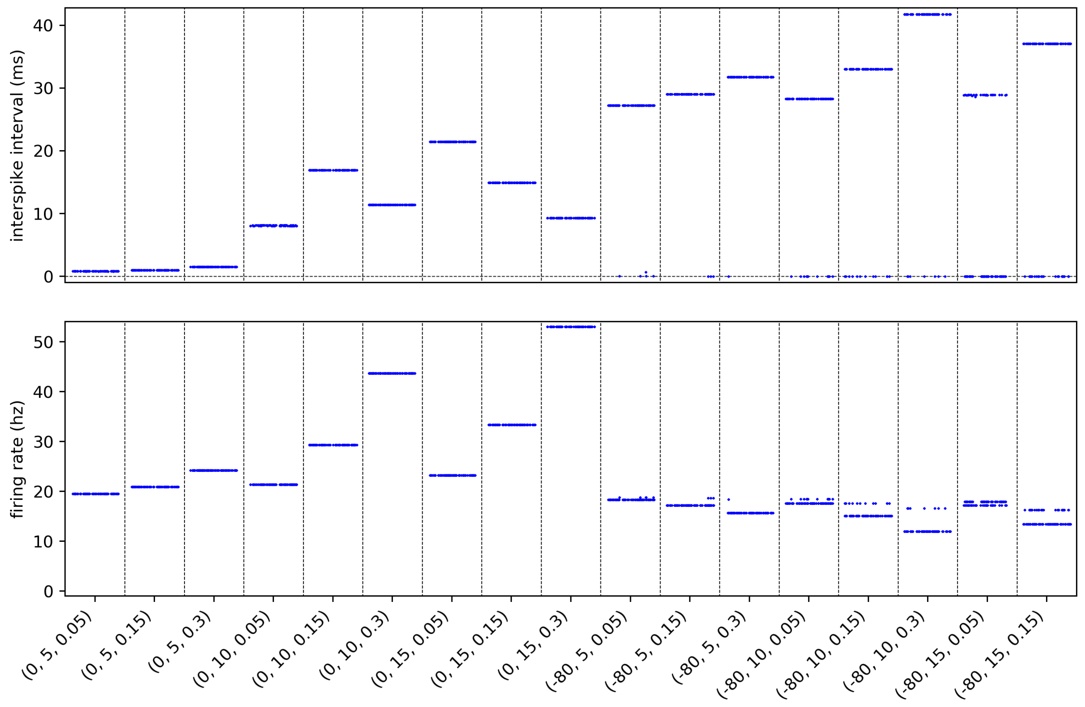


Figure 8: Firing rate (bottom) and interspike interval (top) for 75 simulations at different initial conditions for a range of parameter sets specified using the notation (E_s, τ_s, $r_m\bar{g}_s$). Whenever the dots split into two distinct groups, the long term behavior depends on the initial state of the system with one group corresponding to synchronous firing (interspike interval of zero) and one to alternate firing.

expand the firing rates of each individual neuron, which would also require all M_{AB} to be matrices corresponding to potentially variable connectivity.

The partition into excitatory and inhibitory neurons allows us to easily satisfy Dale's law stating that a presynaptic neuron cannot activate some postsynaptic neurons and inhibit others. This follows from the fact that most characterized neurons release only a single major neurotransmitter, and that a neurotransmitter is generally either excitatory or inhibitory in mammals. Exceptions exist in other organisms, with e.g. acetylcholine having both excitatory and inhibitory receptors in *Drosophila melanogaster*.

We can easily find the nullclines of equations 9 and 10 by setting the lefthand side of each equation equal to zero, which gives

$$v_E = \frac{\gamma_E - M_{EI}v_I}{M_{EE}-1} = \frac{\gamma_E}{M_{EE}-1} - \frac{M_{EI}}{M_{EE}-1}v_I \tag{11}$$

$$v_I = \frac{\gamma_I - M_{IE}v_E}{M_{II}-1} = \frac{\gamma_I}{M_{II}-1} - \frac{M_{IE}}{M_{II}-1}v_E \tag{12}$$

This tells us where the gradient of each firing rate is zero, and these nullclines are plotted in figure 9a. We can also rewrite the expression for v_I nullcline to give v_E as a function of v_I and find the fixed point of this system as the intersection of the two nullclines.

$$v_E = \frac{\gamma_I}{M_{IE}} - \frac{M_{II}-1}{M_{IE}}v_I \tag{13}$$

This gives

$$v_{I,fp} = \left(\frac{\gamma_E}{M_{EE}-1} - \frac{\gamma_I}{M_{IE}}\right)/\left(\frac{M_{EI}}{M_{EE}-1} - \frac{M_{II}-1}{M_{IE}}\right) \tag{14}$$

To determine wheter this fixed point is stable of unstable, we can investigate the eigenvalues of the Jacobian of equations 9 and 10 evaluated at the fixed point, corresponding to the Hessian of (v_E, v_I) which tells us about the behavior of the system near the minimum.

$$J = \begin{bmatrix} \frac{M_{EE}-1}{\tau_E} & \frac{M_{EI}}{\tau_E} \\ \frac{M_{IE}}{\tau_I} & \frac{M_{II}-1}{\tau_I} \end{bmatrix}$$

The eigenvalues of this matrix are given by

$$\lambda = \frac{1}{2}[\frac{M_{EE}-1}{\tau_E} + \frac{M_{II}-1}{\tau_I} \pm \sqrt{(\frac{M_{EE}-1}{\tau_E} + \frac{M_{II}-1}{\tau_I})^2 + \frac{4M_{EI}M_{IE}}{\tau_E\tau_I}}] \tag{15}$$

Armed with these equations, we can begin to characterize our system of interest given a set of physical parameters. In the following we vary τ_I and use fixed parameters given in table 3

M_{EE}	M_{IE}	M_{EI}	M_{II}	γ_E	γ_I	τ_E
1.25	1	-1	-1	$-10Hz$	$10Hz$	$10ms$

Table 3: Fixed parameters in the network model

Together with the equations above, this gives the coordinates of the single fixed point of the system as $v_{I,fp} = 25$ and $v_{E,fp} = 60$.

We further plot the real parts of the eigenvalues of the Jacobian in figure 9b. Dynamical systems theory tells us that the system will be stable if $\Re\lambda_1, \Re\lambda_1 < 0$. The fixed point will be unstable if the real part of at least one eigenvalue is positive. Higher order derivatives must be considered if the real part of the largest eigenvalue is exactly zero.

We start by finding the values of τ_I for which the discriminant of equation 15 is 0, which we do by solving the quadratic equation. This gives

$$\lambda_{D=0} = (8.08, 791.92) \tag{16}$$

We further find that $\Re\lambda_1, \Re\lambda_2 < 0$ for $\tau_I = 8.08$ and $\Re\lambda_1, \Re\lambda_2 > 0$ for $\tau_I = 791.92$. There must therefore be a transition between a stable and unstable fixed point within the interval of $\tau_I \in [8.08, 791.92]$ corresponding to complex eigenvalues. We can find this transition point by ignoring the squareroot term when setting $\Re(\lambda_1), \Re(\lambda_2) = 0$ since this will be imaginary. We thus solve

$$\frac{1}{2}[\frac{M_{EE}-1}{\tau_E} + \frac{M_{II}-1}{\tau_I}] = 0 \tag{17}$$

Which gives

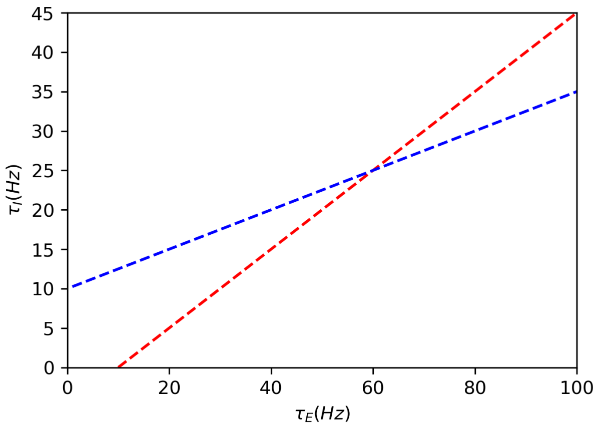
$$\tau_I = \frac{\tau_E(1-M_{II})}{M_{EE}-1} = 80ms \tag{18}$$

This is consistent with the graph in figure 9b which show no other transitions, and we thus conclude that the system has a stable fixed point for all $\tau_I < 80ms$ and an unstable fixed point for all $\tau_I > 80$.

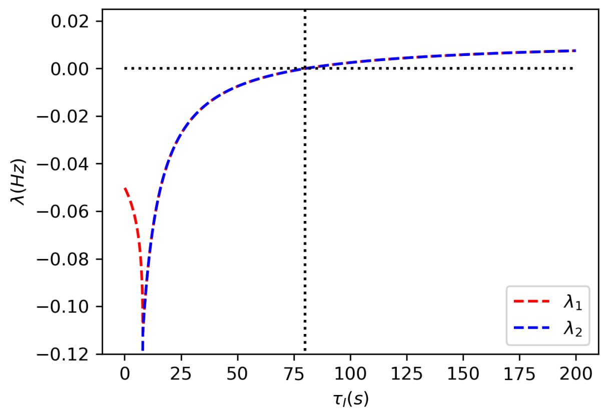
To verify this result, we run a pair of simulations using equations 9 and 10, letting the system evolve over time with $dt = 0.01$. The result of simulations with $\tau_I = 75, 85$ are plotted in figure **??**, projected onto the (v_I, v_E) plane together with the two nullclines as above and a gradient field.

The simulation with $\tau_I = 75$ is initiated at $(v_E, v_I) = (20, 10)$ and we see that it converges to the stable fixed point at $(60, 25)$. For the simulation with $\tau_E = 85$, on the contrary, we start the simulation at $(60, 20)$ near the fixed point, but observe a divergence away from the fixed point as it is unstable in this case.

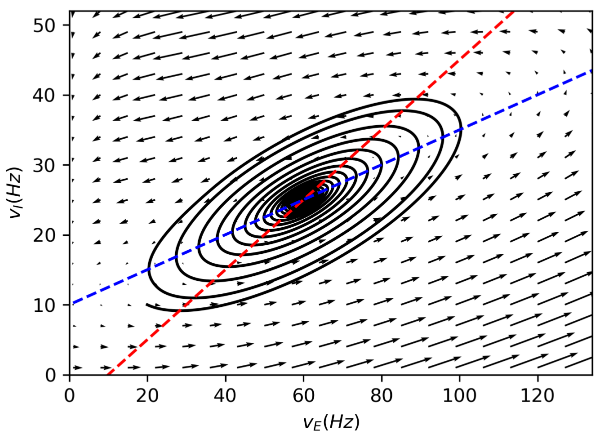
We note that since both simulations have time constants close to $\tau_I = 80$, the gradients near the fixed point are small and convergence is relatively slow near the fixed point for $\tau_I = 75$.



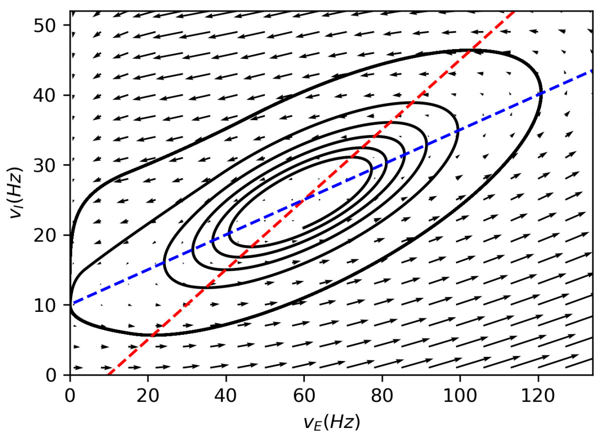
(a) Nullclines for v_E (blue) and v_I (red)



(b) Real parts of the eigenvalues of the Jacobian for the network model as a function of τ_I.



(a)



(b)

Figure 10: Simulation of a network of excitatory and inhibitory neurons with $\tau_I = 75ms$ (left) or $\tau_I = 85ms$ (right). The systems were initialized at $(v_E, v_I) = (20mV, 10mV)$ and $(v_E, v_I) = (60mV, 20mV)$ respectively . We see that the fixed point is stable for $\tau_I = 75ms$ and unstable for $\tau_I = 85ms$

4 Hopfield network

In the following, we investigate the properties of a binary hopfield network. Such a network can be described as a fully connected graph with N nodes and reciprocal connections between every pair of nodes given by a weight matrix W. Each node can take values -1 or 1. We define the energy of a network with states V_i as

$$E = -\sum_{i>j} W_{ij} V_i V_j \tag{19}$$

Given this energy function, we can minimize the energy of the network given a fixed weight matrix for a particular set of initial states by updating the states according to

$$V_i(t+1) = \text{sign}(\sum_{j \neq i} W_{ij} V_j(t)) \tag{20}$$

In the present case we update the network asynchronously but iterate through the nodes systematically until the system is stationary. The change in energy for each step is then

$$\Delta E = (V_i(t) - V_i(t+1)) \sum_{j \neq i} W_{ij} V_j \leq 0 \tag{21}$$

This ensures convergence since the energy is bounded from below.

Mackay and others have often used visual representations to underline the ability or inability of a Hopfield network to recall patterns. However, for a fully connected network, the spatial arrangement of nodes is completely arbitrary, and in the following we therefore pursue a more general approach that is easily quantified and thus allows us to cove a large region of parameter space.

We assess the capacity of a network based on its ability to store randomly generated patterns. We train the network on n_{pat} patterns V^s with states V_i^s by setting the weights according to

$$W_{ij} = \frac{1}{N} \sum_s V_i^s V_j^s \tag{22}$$

i.e. the strength of a connection from i to j is given by the fraction of patterns in which neurons i and j are in the same state minus the fraction of patterns in which their states differ. This can be interpreted as a simple form of Hebbian learning where the strength of a synapse increases the more co-active they are.

To assess the recall of the network, we then select a random sample of 100 learned patterns, for each of these we reverse a fraction f_{err} of the states, and we then compare the pattern converged to from this initial state to the original learned pattern. This corresponds to assessing associative recall given $1 - 2 * f_{err}$ of the original information. The factor of two arises because reversing half the states leads to a complete lack fo correlation whereas reversing all states leads to perfect anticorrelation. We further define an error function between the learned pattern and the converged pattern as

$$\epsilon = 1 - \frac{v_{learned} \cdot v_{converged}}{N} \tag{23}$$

For two random vectors $v_{learned}$ and $v_{converged}$, the expected value of the dot product is zero. Our error function thus in theory runs from 0 to 2 if the two vectors are perfectly anticorrelated, but in practice from 0 to 1 where an error of 1 corresponds to no correlation between the learned and converged patterns.

Given this error function, we start by investigating the performance of assocative recall as a function of the fraction of withheld information f_{err}. We do this by scanning the error landscape over the number of learned patterns and f_{err}, quantifying the error as the mean value of ϵ when training 10 independent networks for a given parameter set and recalling a sample of 100 learned patterns for each network. To begin with, we fix the number of nodes at $N = 300$ as this is not expected to significantly influence the effect of f_{err} for large N. The result is given in figure **??**.

We see that for $f_{err} < 0.12$, the error for a given number of learned patterns only decreases slightly with f_err, with catastrophic forgetting occurring at approximately 50 memorized patterns corresponding to $\frac{n_{pats}}{N} = 0.167$. This is illustrated in figure **??** where the maximum capacity for each f_{err} is plotted. In this case, the capacity is defined as the number of learned patterns for which $\epsilon < 0.20$. Defining the capacity as the point of inflection for a given f_{err} gives a similar but noiser result.

Based on these results, we fix $f_{err} = 0.1$ for the remainder of this report, corresponding to assessing associate recall given 80% of the original data. At $f_{err} > 0.25$, the error rapidly goes to a threshold value of 0.8 even for small numbers of learned patterns suggesting faulty recall given less than 50% of the original data.

Given the methodology described above for assessing the properties of a given Hopfield network, we proceed to investigate how the error rate varies as a function of the number of nodes N and the number of learned patterns (figure 12).

We see that there is a plateau of perfect recall followed by a rapid transition to an asymptotic error of $\epsilon = 0.8$. The reason the error does not go to 1 is that a given query pattern is likely to converge to an energy minimum that is similar to the query, which will in turn also be similar to the original learned pattern. We find that the capacity of the network as a funciton of N follows an approximately straight line ($r = 0.99$) with a slope of $a = 0.151$. This corresponds to a slightly higher capacity than that described by Amit et al (1985), but a qualitatively very similar empirical result to their theoretical considerations yielding a capacity of $n_{pat} = 0.138N$.

We now proceed to vary the sparseness α of the learned patterns with α defined as the probability that a given node is +1 in a pattern. If we query our trained network with a random pattern of sparseness α, we expect the normalized dot product with an uncorrelated pattern of sparseness α to be

$$\tilde{s} = \alpha^2 + (1 - \alpha)^2 - 2\alpha(1 - \alpha) \tag{24}$$

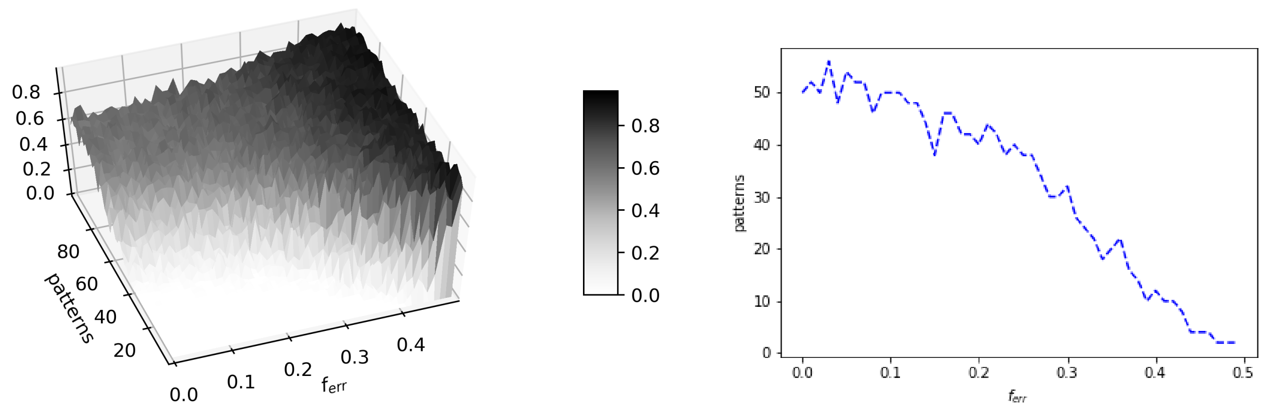


Figure 11: (*left*) error metric ϵ for recall of a given number of learned patterns with a fraction f_{err} of states reversed for a network of N=300 nodes. (*right*) capacity of the network as a function of f_{err}, with the capacity defined as the number of patterns learned before $\epsilon > 0.2$.

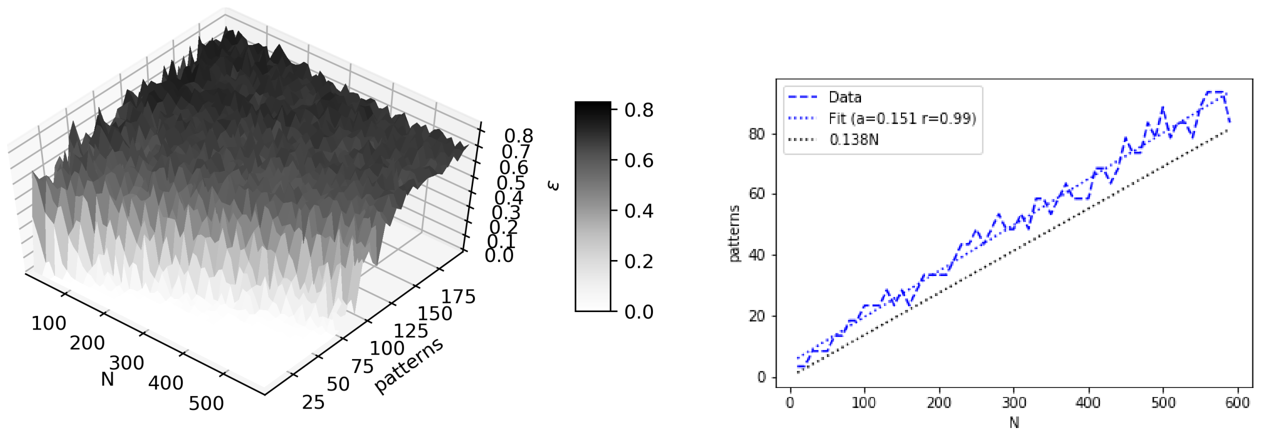


Figure 12: (*left*) error metric ϵ for recall of a given number of learned patterns for hopfield networks of size N with $f_{err} = 0.1$. (*right*) capacity of the network as a function of N, with the capacity defined as the number of patterns learned before $\epsilon > 0.2$. Black dotted line corresponds to the Amit et al. model of a capacity of 0.138N.

For $\alpha = 0.5$, $\tilde{s} = 0$. however for $\alpha > 0.5$, this dot product is greater than zero which should be taken into account in our error function. We therefore define a modified error function as

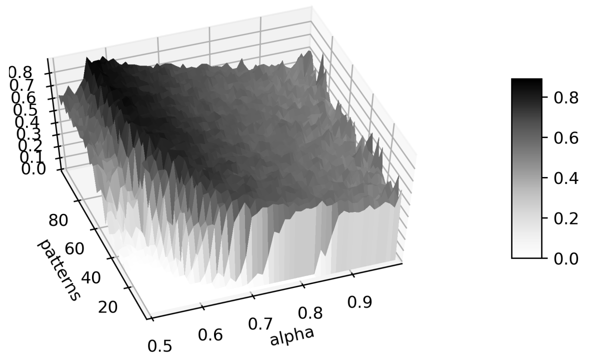
$$\epsilon = \frac{1}{1-\tilde{s}} - \frac{v_{train} \cdot v_{state}}{N(1-s)} \tag{25}$$

Ignoring the effect of introducing a f_{err} errors in our query pattern which slightly reduces the expected value of $\tilde{s}$, this error metric goes from 0 when $v_{train} = v_{state}$ to 1 when they are both random vectors of sparseness α. Note that this expression simplifies to our previous error function when $\alpha = 0.5$ corresponding to random assignment of positive and negative states.

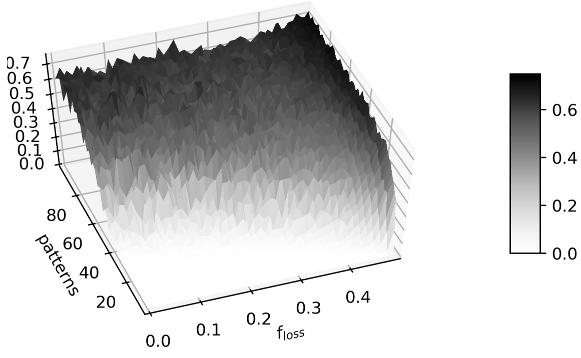
We plot in figure 13a our new error as a function of the sparseness α of the network and the number of patterns learned for N=300. We see that the capacity of the network decays rapidly with α and is effectively 0 at $\alpha = 0.7$. This is because the difference between patterns becomes smaller and minima start coalescing, leading to faulty recall.

We can also invetigate the robustness of the network by setting a random fraction f_{loss} of the weights to zero after training (figure 13b). We see that the network is remarkably robust, retaining 80% of it's capacity even when losing 20% of its weights.

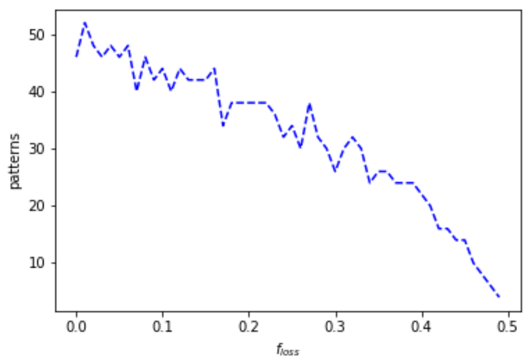
However, it has been previously shown that Hebbian learning does in fact not optimize the storage capacity of our network. Instead, we can train our network by performing steepest descent on an error function of the learned patterns



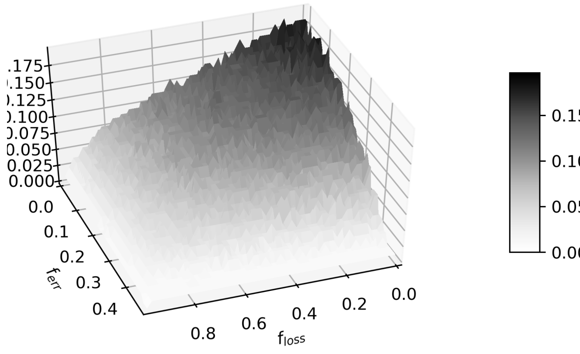
(a) error metric ϵ for recall of a given number of learned patterns for hopfield networks with different sparesness α for N=300 and $f_{err} = 0.1$.



(b) error metric ϵ for recall of a given number of learned patterns for hopfield networks with random loss of a fraction f_{loss} of its weights for N=300 and $f_{err} = 0.1$.



(a) capacity as a function of f_{loss} for $N = 300$ and $f_{err} = 0.1$.



(b) Capacity of the network as a function of f_{loss} and f_{err}.

in a discretized version of the continuous method given in Mackay (algorithm 42.9). This is incidentally also equivalent to the perceptron learning rule introduced in the Deep Learning lectures.

Denoting our input data as X_k^μ for element k of pattern μ, we aim to minimize the error introduced through a single hopfield update step (equation 20). We can find the predicted data for a single Hopfield update step as $T_k^\mu = f(\sum_j X_j^\mu W_{jk}) = \text{sign}(\sum_j X_j^\mu W_{jk})$. This allows us to define an error function

$$E = \sum_{k,\mu} \frac{1}{2}(X_k^\mu - T_k^\mu)^2 \tag{26}$$

This gives

$$\frac{dE}{dW_{ij}} = -\sum_{k,\mu} (X_k^\mu - T_k^\mu) \frac{dT_k^\mu}{dw_{ij}} \tag{27}$$

Of course we cannot easily find the derivative of T_k^μ since we a working with a binary Hopfield network. However we can use a pseudo-gradient assuming the transfer function $f()$ to be linear given the reasoning that increasing the value of $(X^\mu \cdot W_k)$ increases the probability of T_k^μ being 1 rather than -1 and vice versa. We could of course also experiment with other transfer functions, but since the purpose of this section is primarily to exemplify that changing the learning rule can affect Hopfield recall, we consider that beyond the scope of the present report. Assuming a linear transfer function, we get

$$\frac{dT_k^\mu}{dW_{ij}} = \frac{d}{dW_{ij}}(\sum_l X_l^\mu W_{lk}) = X_i^\mu \delta_{jk} \tag{28}$$

this gives

$$\frac{dE}{dW_{ij}} = -\sum_{k,\mu} (X_k^\mu - T_k^\mu) X_i^\mu \delta_{jk} = -\sum_\mu (X_j^\mu - T_j^\mu) X_i^\mu \tag{29}$$

We thus take a steepest descent step with learning rate η at each iteration according to

$$\Delta W_{ij} = \eta \sum_{\mu} (X_j^{\mu} - T_j^{\mu}) X_i^{\mu} \tag{30}$$

Collapsing all of the indices into a set of matrix operations, this is implemented in practice by initializing Hebbian weights and then iterating through the following set of equations until weights are stationary with $\eta = 0.05$ (η was found not to affect the converged weights).

$$T = X \cdot W \tag{31}$$

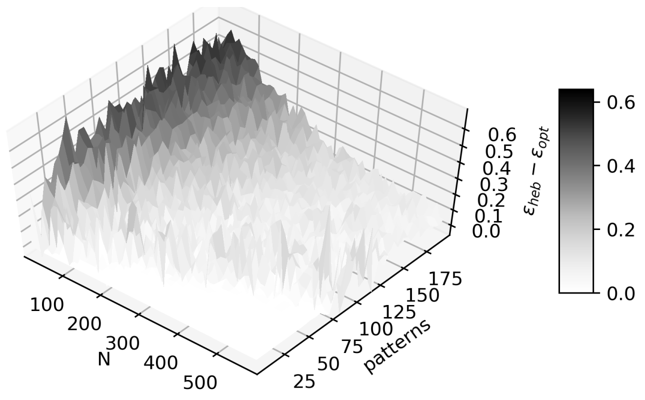
$$E = X - \text{sign}(A) \tag{32}$$

$$gW = X' \cdot E \tag{33}$$

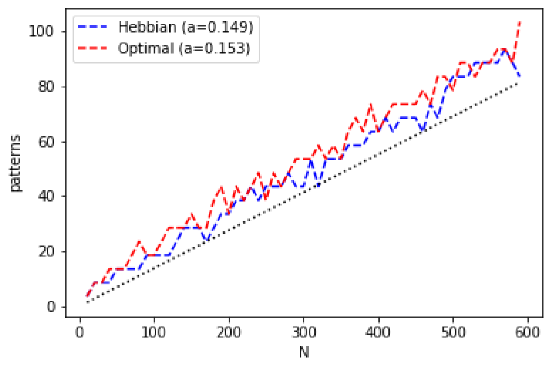
$$W_1 = W_0 + \eta * gW \tag{34}$$

We can now investigate how the performance of the Hopfield network changes as a function of N and n_{pat} for the iteratively optimized weights (opt) compared to the Hebbian weights. We do this by scanning the N, n_{pat} parameter space and plotting the difference in error between the two approaches $\epsilon_{heb} - \epsilon_{opt}$ in figure 15a. We note that this difference is always positive, showing that our optimized weights outperform the Hebbian weights across parameter space. However, we also note that the difference in error is much more substantial at low N than at high N.

We can furthermore observe a narrow 'ridge' in $\Delta\epsilon$ after the $\epsilon = 0$ plateau, suggesting that the optimal network experiences catastrophic forgetting at a slightly higher capacity than the Hebbian network. We therefore plot the capacity as a function of N for both networks in figure **??**. There is indeed a minor difference in capacity, but this generally corresponds to only a small fraction of additional patterns learned, suggesting that while the overall error space is lower for the optimized weights, the near-perfect recall capacity is comparable to the Hebbian network.



(a) Difference in error between Hebbian weights and optimized weights across (N, n_{opt}) parameter space. All values are positive suggesting that the optimized weights always outperform the Hebbian weights.



(b) Capacity of Hebbian (blue) and optimized (red) networks as a function of N. Black dotted line corresponds to the Amit et al. model of a capacity of 0.138N.

Of course this sort of network is only useful because the brain can make sense of the patterns it has stored; i.e. it can decode the information stored in a given pattern of states. Similarly, we can view learned patterns in our Hopfield network as a binary string of length N in which we can encode any information for which autoassociative recall might be desirable.

A simple example of this that we encounter every day is in text autocorrection and autocompletion. We can view the dictionary as a set of stored patterns, and our misspelled word as a faulty query. Any given misspelled word is likely to have only a few typos and we thus often query with more than 70% of the training data, suggesting that this is a task we can use our Hopfield network to solve. Similarly, in an autocompletion process, we might give our network half a sentence and expect it to return the full sentence.

In order to achieve this, we first implement methods for converting an arbitrary lowercase string of length n first to the corresponding 7-digit binary representation for each letter and then to a list of -1 and $+1$ of length $7n$. In this case we require the size of our network to be a multiple of 7, $N = 7i$ where i is an integer. We then add spaces as filling characters to make up for the difference in length between our query string and the capacity of the network since we must train the network with patterns of length N. We also implement a method to decode a given pattern into a string using the inverse methodology.

Given these functions, we can now train out network with a set of strings using the optimized strategy to learn weights since this outperforms Hebbian learning. As an example of autocorrection, we construct a net of size $N = 84$ corresponding to a maximum if 12 characters. We then train it on the three strings ['hopfield', 'onomatopoeia', 'accommodate'].

Now when we try to write our report on hopfield networks, but accidentally think they were developed by John Ho**b**field, we get
network.query_string('hobfield') → 'hopfield'
Similarly, when we forget how to spell onomatopoeia, we get
network.query_string('onomatopeia') → 'onomatopoeia'

However, this implementation is strictly dependent on exact character matching in terms of location. Thus when we forget that 'accommodate' has two c's and two m's, all of the subsequent characters are mismatched and we fail to recover our original word
network.query_string('acommodate') → 'iocoieodedia'

Similarly, we can use a hopfield network for sentence autocompletion if we teach it a set of sentences. In the present case, we initialize a network with $N = 235$ corresponding to 35 characters and learn weights for the following set of strings: ['i hope stephen gives me a good mark', 'i really like chocolate mousse', 'i really like strawberries with milk']. We then query the network with a partial sentence:
network.query_string('i hope stephen') → 'i hope stephen gives me a good mark'

However, in the case of ambiguity, the network tends to fail and give a mixed result:
network.query_string('i really like') → 'i really like sdkcgbete ewuwkdh ijmk'

And as above, we are dependent on strict positional character matching:
network.query_string('strawberries with milk') → 'i really like chocolate mousse'
network.query_string('   strawberries with milk') → 'i really like strawberries with milk'

Of course noone would actually use a Hopfiel network for autocompletion and autocorrection in practice as there are many smarter algorithm in place that can also take into account contextual information and which have much higher capacities and superior recall. However, the above serves as an illustration that such networks are not limited to random patterns and visual representations, since this would be of little use to the brain. Instead they can be used for autoassociative recall of any information that can be encoded and decoded as binary strings.

Appendix

```
using PyCall, PyPlot, LinearAlgebra, Random, DelimitedFiles


function get_ints(spikes)
    intervals = []
    rates = []
    if spikes[1][1] < spikes[2][1]
        j = 1
    else
        j = 2
    end
    for i in 1:(length(spikes[j])-1)
        intervals = [intervals; spikes[3-j][i]-spikes[j][i]]
        rates = [rates; 1/(spikes[j][i+1]-spikes[j][i])*1000]
    end
    return [spikes[j][1:length(intervals)], intervals, rates]
end

function plotIF(times, V, spikes; interval=true, filename="test")

    figure()
    plot(times, V[:,1], "b-")
    plot(times, V[:,2], "r-")
    types = ["b-", "r-"]
    for j in 1:2
        for s in spikes[j]
            plot([s,s], [-80,0], types[j])
        end
    end
    xlim(times[1], times[end])
    xlabel("t␣(ms)")
    ylabel("V␣(mV)")
    savefig(filename*".png")
    show()
    close()

    if interval
        times, intervals, rates = get_ints(spikes)
        figure()
        plot(times, intervals)
        savefig(filename*"_int.png")
        close()

        return [times, intervals]
    end
end

function IF(V1, V2; Es=0, T=2000, dt=0.005, Pmax=0.5, ts=10, tm=20, rmgs=0.15,
            RmIet=18, EL=-70, Vt=-54, Vreset=-80, plotrange="all", filename="test")

    N = Int(round(T / dt) + 1)
    times = 0:dt:T
    V = zeros(N,2)
    Ps = zeros(N,2)
    z = zeros(N, 2)
    V[1,:] = [V1; V2]
    spikes = [ [], [] ]
    for i in 2:N
        for j in 1:2
            z[i,j] = z[i-1,j] - dt/ts * z[i-1,j]
            Ps[i,j] = Ps[i-1,j] + dt/ts * (exp(1) * Pmax*z[i,j]-Ps[i-1,j])
            V[i,j] = V[i-1, j] + dt/tm *
                (EL - V[i-1,j] -rmgs*Ps[i-1,j]*(V[i-1, j]-Es)+RmIe)
        end
        for j in 1:2
            if V[i,j] >= Vt
                z[i, 3-j] = 1
                V[i,j] = Vreset
                spikes[j] = [spikes[j];times[i]]
            end
        end
    end

    if plotrange == "none"
        a = 2
    elseif plotrange == "all"
        plotIF(times, V, spikes, filename=filename)
    else
        plotrange = Int(plotrange[1]/dt):1:Int(plotrange[2]/dt)
        plotIF(times[plotrange], V[plotrange,:], spikes, filename=filename)
    end

    return times, V, spikes, z, Ps

end

function test_intervals(;ts=10, Es=-80, rmgs = 0.15)
    ints = [[], [], [], [], [], []]
    if Es == -80
        Vis = [-71; -72; -72.35; -72.4; -73; -74]
    else
        Vis = [-70.5; -72; -74; -76; -78; -80]
    end
    for i in 1:length(Vis)
        times, V, spikes, z, Ps = IF(-70, Vis[i], Es=Es, T=3000,
            plotrange="none", ts=ts, rmgs=rmgs)
        ints[i] = plotIF(times, V, spikes)
    end
    figure()
    cols = ["b--", "r--", "g--", "y--", "c--", "m--"]
    for i in 1:length(Vis)
        plot(ints[i][1], ints[i][2])
    end
    xlabel("t (ms)")
    ylabel("interspike interval (ms)")
    legend([string(Vi) for Vi in Vis])
    savefig("intervals_ts_"*string(ts)*"_rmgs_"*string(rmgs)*"e.png")
    show()
    close()
end

function random_init(;N=75)#Es, ts, rmgs; N=5)

    Ess = [0, -80]
    tss = [5, 10, 15]
    rmgss = [0.05, 0.15, 0.30]

    n = 0
    labs = []
    ncats = length(Ess)*length(tss)*length(rmgss)-1
    ints = zeros(N, ncats)
    rates = zeros(N, ncats)
    xs = zeros(N, ncats)
    for Es in Ess
        for ts in tss
            for rmgs in rmgss
                if !(Es==-80 && ts == 15 && rmgs == 0.30) #no spikes
                    println("new params ", Es, " ", ts, " ", rmgs)
                    labs = [labs; "("*string(Es)*",␣"*string(ts)*",␣"*string(rmgs)*")"]
                    n += 1
                    #ints = []
                    #rates = []
                    for i in 1:N
                        println("new␣i", i)
                        v1, v2 = rand(2)*25-[80,80]
                        times, V, spikes, z, Ps = IF(v1, v2, Es=Es, T=10000,
                            plotrange="none", ts=ts, rmgs=rmgs)
                        res = get_ints(spikes)
                        is = res[2]; rs = res[3]
                        #rates = [rates; rs[end]]
                        rates[i,n] = rs[end]
                        if is[end] <= (1/rs[end])*1000/2
                            ints[i,n] = is[end]
                        else
                            ints[i,n] = (1/rs[end])*1000 - is[end]
                        end
                    end
                    xs[:,n] = rand(N)*0.8 + repeat([n-0.40], N)
                    #plot(xs, ints, "bx", MarkerSize = 1)
                end
            end
        end
    end

    fignames = ["ints_scatter", "rates_scatter"]
    ylabs = ["interspike␣interval␣(ms)", "firing␣rate␣(hz)"]
    for i in 1:2
        data = [ints, rates][i]
        figure(figsize = [11,3])
        if i == 1
            plot([0.5, ncats+0.5], [0,0], linestyle="--",color="0.1", linewidth = 0.5)
        end
        for n in 1:ncats
            plot(xs[:,n], data[:,n], "bx", MarkerSize = 1)
            plot([n+0.5, n+0.5], [-10, 100], "k--", linewidth = 0.5 )
        end
        if i == 2
            xticks(1:length(labs), labs, rotation=45, ha="right")
        else
            xticks([])
        end
        ylim([-1, maximum(data)+1])
        xlim(0.5, ncats+0.5)
        ylabel(ylabs[i])

        savefig(fignames[i]*".png", bbox_inches="tight", dpi=360)
        show()
        close()
    end
    #writedlm("ints.dlm", ints)
    #writedlm("rates.dlm", rates)
    #writedlm("xs.dlm", xs)
end

#random_init()#-80, 10, 0.15)
```

```
test_intervals(Es=0, rmgs=0.5, ts=10)
#times, V, spikes, z, Ps = IF(-65, -75,
#     Es=0, T=3001, rmgs=0.05, ts=5, plotrange=[2500,3000], filename="test")



# #potentials in mV
# EL = -70 #equlibrium leak potential
# Vt = -54 #spike threshold
# Vreset = -80 #refractory potential
#
# #times in ms
# tm = 20 #membrane time constant
# rmgs = 0.15 #synaptic conductivity times membrane resistance. units cancel
# RmIe = 18 #mV. external input current
#
# T = 500
# dt = 0.005
#ts = 10
#Pmax = 0.5

#dPs = dt / ts * ( exp(1) * Pmax*z-Ps )
#dz =- dt / ts * z

#Es = 0
```