Computational Neuroscience: Assignment 1

University of Cambridge

Henrik Åhl

April 21, 2017

Preface

This is an assignment report in connection to the *Computational Neuroscience* module in the Computational Biology course at the University of Cambridge, Lent term 2017. All related code is as of April 21, 2017 available through a Github repository by contacting hpa22@cam.ac.uk.

Exercises

1 The four parameters v, u, I_e and ϵ denote the neuron membrane voltage, a recovery variable, an external stimulus input current, and a time-scale parameter respectively. We plot the nullclines in fig. 1. Implementing Newton's method to find the roots of the composite equation $v^3 + v + 1 = 0$, we find that they are given by $0.5u = v$ = -0.6823278, -0.6823278i and 0.6823278i for $I_e = -1$, and $u = v = 0$ and $v = \pm i, u = \pm 2i$ for $I_e = 0$. Observing the nullcline figures, we see that this appears visually likely as well.

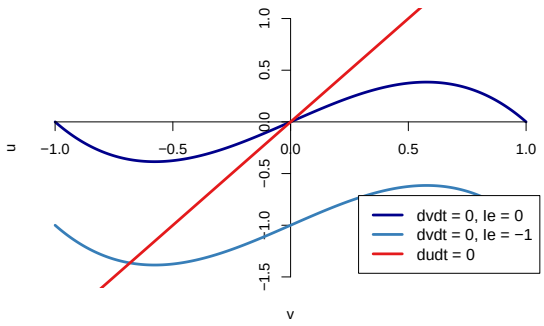

Figure 1: Nullclines of the system plotted as functions of u.

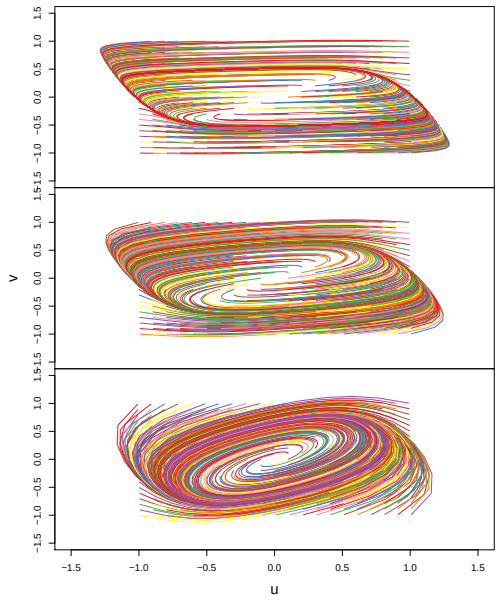

Figure 2: Trajectories for $\epsilon = \{0.1, 0.3, 1.0\}$ respectively, with $I_e = 0$. Note how all trajectories diverge from the unstable fixpoint at the origin, and how the eccentricity of the phase diagram changes with altered ϵ.

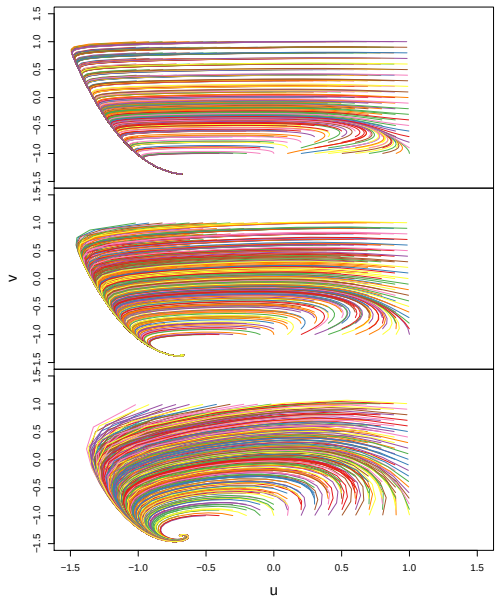

Figure 3: Similar to fig. 2, but for $I_e = -1$. Note how all trajectories converge to the same point, and again, how the eccentricity changes with ϵ.

When analysing the stability of the system, we consider the Jacobian matrix we get from our equations:

$$\mathbf{J} = \begin{pmatrix} 1 - 3v^2 & -1 \\ \epsilon & \frac{\epsilon}{2} \end{pmatrix} \tag{1.1}$$

For stability we require, as before, that

$$0 = v(1 - v^2) - u + I_e \tag{1.2}$$

$$0 = \epsilon(v - \frac{u}{2}) \tag{1.3}$$

as previously, which again gives us $u = 2v$, but also $I_e = v(1 + v^2)$.

From the Hopf bifurcation theorem, we require $Tr(\mathbf{J}) = 0$, and as a results also $det(\mathbf{J}) > 0$. Investigating this we get that

$$Tr(\mathbf{J}) = 1 - 3v^2 - \frac{\epsilon}{2} = 0 \Leftrightarrow 1 - 3v^2 = \frac{\epsilon}{2} \tag{1.4}$$

$$\Rightarrow v = \pm\sqrt{\frac{1 - \frac{\epsilon}{2}}{3}} \tag{1.5}$$

$$det(\mathbf{J}) = \epsilon - \frac{\epsilon}{2}(1 - 3v^2) > 0 \tag{1.6}$$

from where it follows that

$$\epsilon(\epsilon + 3) > 0 \Rightarrow \epsilon < -3 \vee \epsilon > 0. \tag{1.7}$$

Imposing the physical requirement of positive ϵ, we see that we simply require the constant to be non-zero. Considering I_e we find from above that

$$I_e(\epsilon) \lessgtr \pm\sqrt{\frac{1 - \frac{\epsilon}{2}}{3}}\left(\frac{4}{3} - \frac{\epsilon}{6}\right) \tag{1.8}$$

where we also see that we require $\epsilon \leq 2$ for a real-valued solution. Plotting this region, we get the one seen in fig. 4.

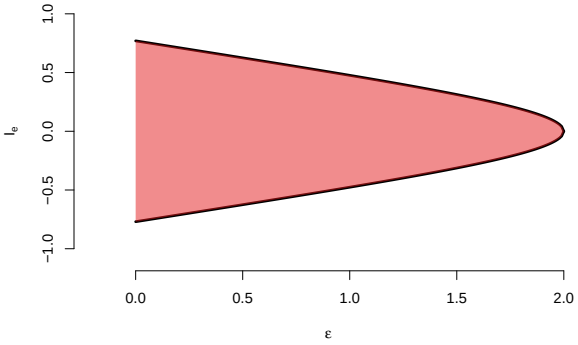

Figure 4: Parameter area giving rise to oscillation in thr system. Note how we indeed for all cases when $I_e = 0$ can expect oscillations for our choices of ϵ, but for none of the cases when $I_e = -1$.

2 Using a perceptron with a given bias node, we implement the three rules to classify our input data. In all our comparisons we use 10 inputs (and weights) aside of the bias. When not comparing learning time, we iterate the perceptron and delta rules 10 times the number of patterns contained in the training data set. We also consistently use a learning rate of 0.1 and replicate each setting 200 times for statistical purposes.

When using a linear input function for the delta rule, it reduces to the perceptron rule. We therefore instead of a linear input function modify our inputs by parsing them through the error function, which simply increases the sign sensitivity of the data. However, the effect this gives is to simply this only corresponds to changing the learning rate per iteration and therefore does not affect the overall dynamics much, as we can see in fig. 5. All rules otherwise perform well in the simulations when summation classification is used on the output. When multiplicative classification is used, the performance is overall very scattered, and the network often does not perform better than random, showing immense troubles in classifying the

data. This is because the product version is not linearly separable (compare to `XOR` in two dimensions). Because of this, the rule will also tend to not converge, and we are therefore unable to provide estimates for the learning rate. Resultingly, the only such graph we can produce is seen in fig. 6, where a comparison between the delta and perceptron rule is seen. As expected, due to the similarities, the rules perform comparably.

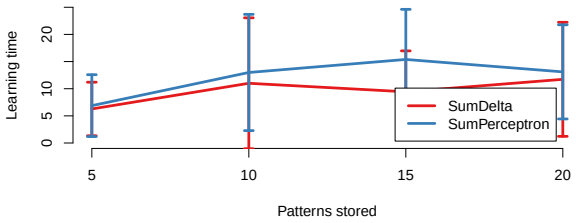

Figure 6: Learning times for the delta and perceptron rule under the summation method for a small range of patterns stored in a 10 node perceptron. Due to the high similarities, the rules perform equivalently. The time here is measured as the number of iterations until all input patterns are recalled correctly.

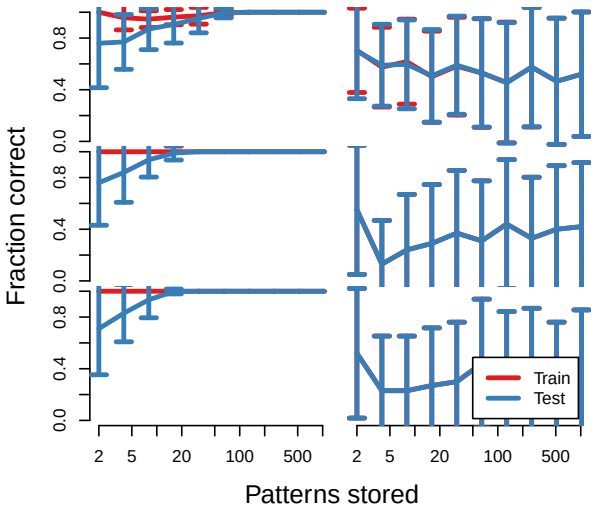

Figure 5: Performance of pattern recall for select number of patterns, in a perceptron consisting of 10 weights. The summarisation method is capable of storing nets efficiently, with the hebbian rule performing generally on par with the others, although it is trending towards performing slightly worse when the number of patterns stored is less then the number of nodes. The delta rule with the error input function performs effectivelly equivalently to the perceptron rule as the weights are able to easily adapted regardless of this. The product method performs horribly for all cases, emphasising the linear inseparability that makes the classification difficult.

3 Figure 7 shows the orientation of the weight vector under the correlation and covariance rule in the multiplicative and subtractive case, with exemplary randomly assigned initial weights. While it appears to be the case for the correlation method to align to the first principal component of the data, deviating from the mean shows that this is not the case. Further investigations show that it instead aligns to the first eigenvector of the correlation matrix. In contrast, the covariance matrix will by definition have eigenvectors corresponding to the principal components of the data (since the principal components give the vectors with highest variance), and the weight vector will therefore evolve in accordance to that, eventually reaching the stable state of being aligned.

For the subtractive normalisation with boundaries, the weight vector will evolve according to the eigenvectors of the correlation/covariance matrix minus the average weight change. This will effectively force one weight to steadily increase while the other decreases, meaning that the weights will evolve until on of them hits the boundaries. Consequently, if no action is taken to stop the change for one of the weights while the other hits the boundary, the vector will evolve until the two weights are at either bound, i.e. 0 and 1 in our case.

In fig. 8, we can see the final outcome of the simulations based on the initial conditions. For the multiplicative case, the product of the weights are shown, while we in the subtractive case see only where w_1 is when any of the weights saturate. We note that we in the multiplicative case clearly have stable fixpoint in aligning to the first PC in most of the cases, but another attractor when the sum of the weights are close to 0. This ought to be because there is an effective division with respect to the sum of the

weights, which overthrows the natural development of the weights if causing a large enough shift in relation to the non-diagonality of the correlation matrix. In the covariance case, the difference between the diagonal elements is too small to be noticeable, and we only see the effect when the sum is indeed 0.

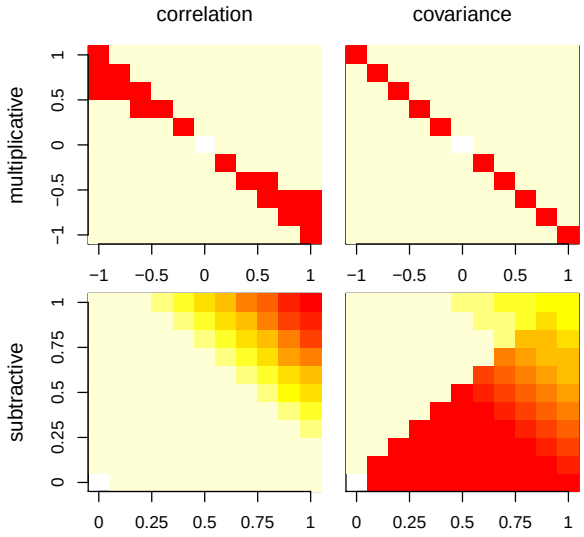

Figure 8: Outcome dependence on initial conditions when using the correlation method, $x_{mean} = 10$. The axes represent the initial weights, whereas the colours in the multiplicative case represent the value of the product of the weights, and in the subtractive case simply the value of w_1. That is, in the multiplicative case, we get information of which eigenvector of the correlation/covariance matrix the weights have aligned to, while we in the subtractive case simply get information of where the first weight is when the second weight hit a boundary. Under zero mean, the effects will look similar, but the correlation and the covariance case functionally equivalent under multiplicative normalisation, i.e. both with a hard boundary.

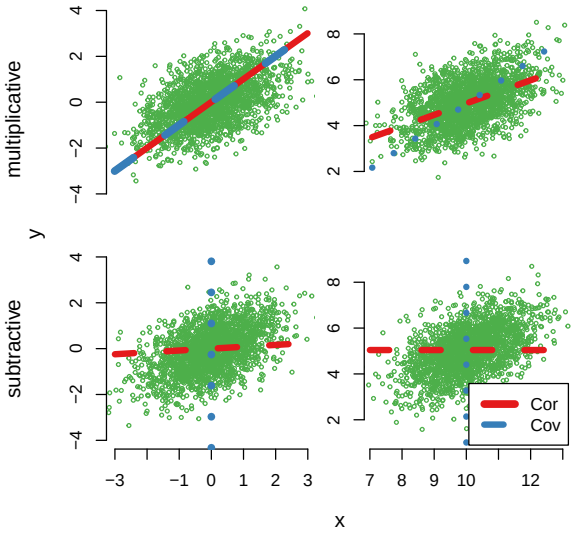

Figure 7: Final weight vector on top of the data under zero and non-zero circumstances, under both normalisation conditions. Weights initialised randomly. Note how the weight vectors appear to align with the first principal component in the multiplicative case for zero mean.5 Under non-zero mean we do however see how the correlation method aligns to the first eigenvector of the correlation matrix, rather than the principal component of the data. The covariance matrix is nonetheless able to align to the data. Under some circumstances, the vectors align to the second eigenvector (see fig. 8).

Figure 9 shows weight trajectories for most of the interesting cases in fig. 8. Wee see in particular how the non-diagonality of the matrices cause different behaviours depending on which weight starts with what value. Even though the ultimate alignment is the same, the weight dynamics are different. The ability of the vectors to align to the principal components will thus depend on the eigenvectors of the matrix governing the updates and the values of the initial weights. Under subtractive normalisation, the weights will only align under *very* improbable conditions.

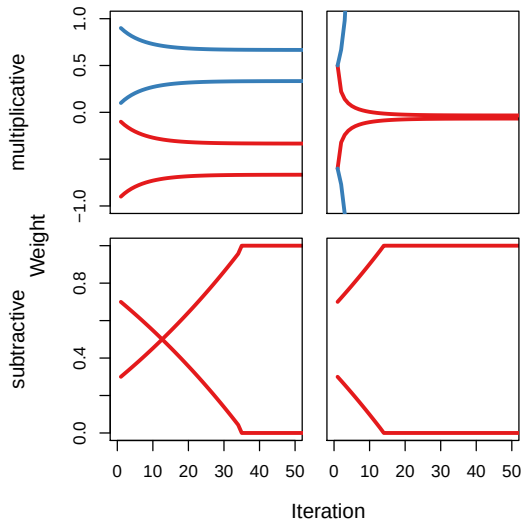

Figure 9: Weight development trajectories for exemplary initial values, corresponding to the different colours in fig. 8. The colours in this figure separates different initial conditions within the same figure. Note how which weight being what changes the dynamics drastically in the multiplicative case when they are of opposite sign, and in the subtractive case when they are of different magnitude.

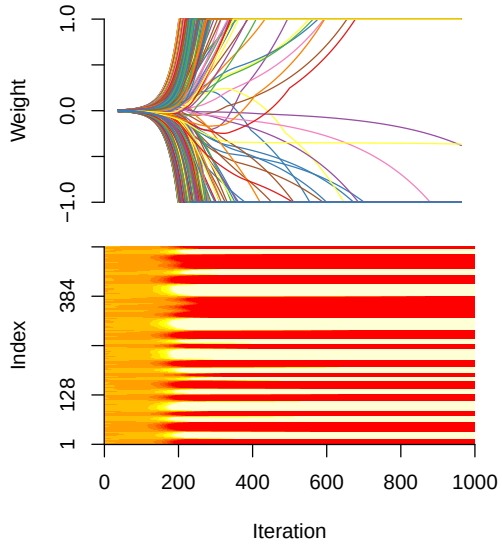

Figure 10: Ocular dominance patterns driven by weight saturation under subtractive normalisation. The upper figure shows the weight trajectories (w_-) for individual cells, whereas the latter shows the same on a map display.

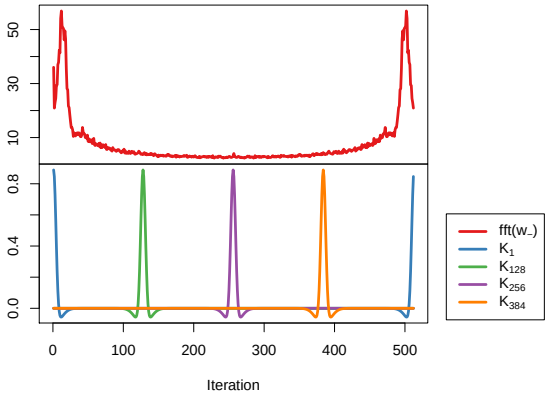

Figure 11: Upper: Fourier transform of the means of the final w_-, over 100 simulations. Lower: Individual vectors of the interaction matrix K. Note the slightly negative values close to the peaks, showing local competition.

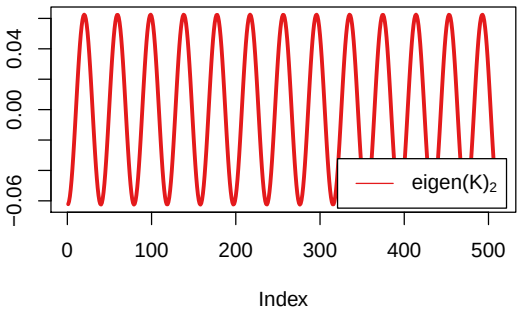

Figure 12: Exemplary K eigenvector, showing how the eigenvectors of K correspond to cosinusoidal functions.

4 Letting the weights be allowed to move freely within the boundaries, we can see how the weights develop in fig. 10, where we in the lower part can notice the arisal of the characteristic stripes of ocular dominance. By changing sigma, i.e. the inter-neuronal interaction, we can affect the stripe width as we like. We have removed edge-effects by setting neuron 512 and neuron 1 to be adjacent, and imposing the distance metric under this toroidal landscape.

Figure 11 also shows the mean of the Fourier transforms of the final weights, along with some select rows of the neuronal interaction matrix. The eigenvectors of K will be cosinusoidal functions, which we easily see for example value K_2 in fig. 12. Similarly, the eigenvalues give the corresponding frequencies.

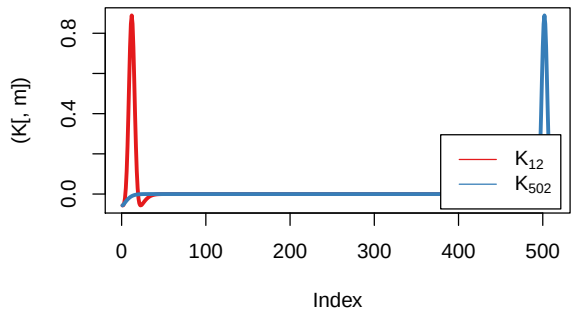

Figure 13: The two principal eigenvectors driving the system, first in red, second in blue. Indices are 12 and 502 respectively.

We also see that the mean of the Fourier transform of the final w_- develops according to the eigenvectors maximizing the Fourier transform of K, which we can see by taking the index maximizing the mean Fourier transform of the weights and plotting it, as in fig. 13, where the red curve corresponds to the first PC, and the blue the second.

2 Acknowledgements

As always, thanks to Julian Melgar for no particular reason. Also thanks to Olena Yavorska for proving that tidyness trumps compactness.

A Code

```
#!/usr/bin/env Rscript
setwd("~/compbio/src/assignments/cnsa2/code/")
library(RColorBrewer)
colours = brewer.pal(n = 8, name = "Set1")
palette(colours)

# Dynamical parameters
method = "euler"
no.replicates = 1
epsilon = NA
Ie = NA
start.v = NA # runif(1, max = 1, min = -1)
start.u = NA # runif(1, max = 1, min = -1)

# Simulation parameters
t = NA
start.t = 0.0
end.t = 500.0
no.outputs =  (end.t - start.t)*4 # Print four times every time unit
h = 0.01
no.iterations = (end.t - start.t) / h
output = data.frame(v = rep(NA, no.outputs), u = rep(NA, no.outputs))
printpoints = seq(start.t, end.t, length.out = no.outputs)
rownames(output) = printpoints

# Define derivatives
v.derivs = function(t, v, u) {
  v * (1 - v ** 2) - u + Ie
}

u.derivs = function(t, v, u) {
  epsilon * (v - 0.5 * u)
}

# Simple Euler integration
euler = function(t, v, u, h, ...) {
  v.k1 = v.derivs(t, v, u)
  u.k1 = u.derivs(t, v, u)
  v <<- v + v.k1 * h
  u <<- u + u.k1 * h
  t <<- t + h
}

# Fourth-order Runge-Kutta
rk4 = function(t, v, u, h, vol) {
  v.k1 = v.derivs(t, v, u)
  u.k1 = u.derivs(t, v, u)
  v.k2 = v.derivs(t + h / 2.0, v + h / 2.0 * v.k1, u + h / 2.0 * u.k1)
  u.k2 = u.derivs(t + h / 2.0, v + h / 2.0 * v.k1, u + h / 2.0 * u.k1)
  v.k3 = v.derivs(t + h / 2.0, v + h / 2.0 * v.k2, u + h / 2.0 * u.k2)
  u.k3 = u.derivs(t + h / 2.0, v + h / 2.0 * v.k2, u + h / 2.0 * u.k2)
  v.k4 = v.derivs(t + h, v + h * v.k3, u + h * u.k3)
  u.k4 = u.derivs(t + h, v + h * v.k3, u + h * u.k3)
  v <<- v + h / 6.0 * (v.k1 + 2 * v.k2 + 2 * v.k3 + v.k4)
  u <<- u + h / 6.0 * (u.k1 + 2 * u.k2 + 2 * u.k3 + u.k4)
  t <<- t + h
}

# Set numerical method
if (tolower(method) == "milstein") {
  fct = milstein
} else if (tolower(method) == "euler") {
  fct = euler
} else if (tolower(method) == "rk4") {
  fct = rk4
```

```
} else {
  stop("Error: Method not found.")
}


vs = seq(-1, 1, 0.1) # remember to change back
us = seq(-1, 1, 0.1)

runSimulation = function(v.init, u.init, epsilon, Ie) {
  # initialise(v.init, u.init, epsilon, Ie)
  v <<- v.init
  u <<- u.init
  epsilon <<- epsilon
  Ie <<- Ie
  t <<- start.t
  t.out <<- 1
  output[1,] <<- c(v, u)

  for (iter in 1:(no.iterations)) {
    fct(t, v, u, h, vol)
    if (t >= printpoints[t.out]) {
      output[t.out, ] = c(v, u)
      t.out = t.out + 1
    }
  }
  return(output)
}
# Run single-valued simulations
# results = replicate(no.replicates, runSimulation(start.v, start.u, epsilon, Ie))

get.period = function(data){
  # Get minima
  indices = which(data[2:(length(data)-1)] < data[3:length(data)] & data[2:(nrow(data)-1)] <
      data[1:(length(data)-2)])
  # times = indices + 1 # +1 due to offset
  times = printpoints # +1 due to offset
  periods = times[2:length(times)] - times[1:(length(times)-1)]
  return(periods)
}

# With I_e = 0
 results.1 = lapply(us, function(u) lapply(vs, function(v) runSimulation(u, v, 0.1, 0)))
 results.2 = lapply(us, function(u) lapply(vs, function(v) runSimulation(u, v, 0.3, 0)))
 results.3 = lapply(us, function(u) lapply(vs, function(v) runSimulation(u, v, 1.0, 0)))
#load("../data/trajectories.RData")
# par(
#   mfrow = c(3, 1),
#   oma = c(10, 5, 4, 2) + 0.0,
#   mai = c(.0, .6, .0, .5)
# )
# plot(1, xlim = c(-1.5,1.5), ylim = c(-1.5,1.5), cex = 0, xaxt = "n", ylab="")
# tmp = sapply(1:length(us), function(x) sapply(1:length(vs), function(y)  lines(results.1[[x
    ]][[y]][,1], results.1[[x]][[y]][,2], col = sample(colours))))
# plot(1, xlim = c(-1.5,1.5), ylim = c(-1.5,1.5), cex = 0, xaxt = "n", ylab="")
# mtext("v", side=2, line=3,las=3, col="black", cex = 1)
# tmp = sapply(1:length(us), function(x) sapply(1:length(vs), function(y)  lines(results.2[[x
    ]][[y]][,1], results.2[[x]][[y]][,2], col = sample(colours))))
# plot(1, xlim = c(-1.5,1.5), ylim = c(-1.5,1.5), cex = 0, ylab="")
# tmp = sapply(1:length(us), function(x) sapply(1:length(vs), function(y)  lines(results.3[[x
    ]][[y]][,1], results.3[[x]][[y]][,2], col = sample(colours))))
# mtext("u", side=1, line=3,las=1, col="black", cex = 1)
#
# With I_e = -1
 results.1.2 = lapply(us, function(u) lapply(vs, function(v) runSimulation(u, v, 0.1, -1)))
 results.2.2 = lapply(us, function(u) lapply(vs, function(v) runSimulation(u, v, 0.3, -1)))
 results.3.2 = lapply(us, function(u) lapply(vs, function(v) runSimulation(u, v, 1.0, -1)))
 save(results.1, results.2, results.3, results.1.2, results.2.2, results.3.2, file = "../data
```

```
      /trajectories.RData")
129
130 # plot(1, xlim = c(-1.5,1.5), ylim = c(-1.5,1.5), cex = 0, xaxt = "n", ylab="")
131 # tmp = sapply(1:length(us), function(x) sapply(1:length(vs), function(y)  lines(results
      .1.2[[x]][[y]][,1], results.1.2[[x]][[y]][,2], col = sample(colours))))
132 # plot(1, xlim = c(-1.5,1.5), ylim = c(-1.5,1.5), cex = 0, xaxt = "n", ylab="")
133 # mtext("v", side=2, line=3,las=3, col="black", cex = 1)
134 # tmp = sapply(1:length(us), function(x) sapply(1:length(vs), function(y)  lines(results
      .2.2[[x]][[y]][,1], results.2.2[[x]][[y]][,2], col = sample(colours))))
135 # plot(1, xlim = c(-1.5,1.5), ylim = c(-1.5,1.5), cex = 0, ylab="")
136 # tmp = sapply(1:length(us), function(x) sapply(1:length(vs), function(y)  lines(results
      .3.2[[x]][[y]][,1], results.3.2[[x]][[y]][,2], col = sample(colours))))
137 # mtext("u", side=1, line=3,las=1, col="black", cex = 1)
138
139 # # Run simulations for multiple parameters
140 # start.vs = seq(0,1,.25)
141 # start.vs = c(start.vs, seq(0,1,.25))
142 # start.vs = c(start.vs, seq(0,1,.25))
143 # start.us = seq(0,1,.25)
144 # start.us = c(start.us, seq(0,1,.25))
145 # start.us = c(start.us, seq(0,1,.25))
146 # epsilons = c(rep(0.1, length(start.vs) / 3), rep(0.3, length(start.vs) / 3), rep(1, length(
      start.vs) / 3))
147 # results = sapply(1:no.replicates, function(x) runSimulation(start.vs[x], start.us[x],
      epsilons[x], Ies[x]))
148
149 # results = replicate(no.replicates, runSimulation(start.v, start.u, .1, 0))
150 # results.2 = replicate(no.replicates, runSimulation(start.v, start.u, .3, 0))
151 # results.3 = replicate(no.replicates, runSimulation(start.v, start.u, 1, 0))
152 # par(
153 #   mfrow = c(3, 1),
154 #   oma = c(10, 5, 4, 2) + 0.0,
155 #   mai = c(.0, .6, .0, .5)
156 # )
157 # linewidth = 2.5
158 # plot(seq(start.t,end.t - 1/print.ps, 1 / print.ps), results[,1]$u, type = "l", xaxt = "n",
      xlab = "n", ylab = "u", lwd = linewidth, col = 1) # Plot u
159 # plot(seq(start.t,end.t - 1/print.ps, 1 / print.ps), results[,1]$v, type = "l", xaxt = "n",
      xlab = "n", ylab = "v", lwd = linewidth, col = 2) # Plot v
160 # plot(results[,1]$u, results[,1]$v, type = "l", ylab = "v", xlab = "u", lwd = linewidth, col
      = 1, xaxt = "n") # Plot u vs v
161 # plot(results[,1]$u, results[,1]$v, type = "l", ylab = "v", xlab = "u", lwd = linewidth, col
      = 1, xlim = c(-1.1,1.1), ylim = c(-1.1,1.1), xaxt = "n") # Plot u vs v
162 # legend("topleft", c(expression(epsilon == 0.1), expression(epsilon == 0.3), expression(
      epsilon == 1.0)), col = c(1,2,3), inset = c(0.01, 0.05), lty = 1, lwd = linewidth, bg="
      white", cex = 1.3, box.lwd = 0)
163 # plot(results.2[,1]$u, results.2[,1]$v, type = "l", ylab = "v", xlab = "u", lwd = linewidth,
       col = 2, xaxt = "n") # Plot u vs v
164 # plot(results.3[,1]$u, results.3[,1]$v, type = "l", ylab = "v", xlab = "u", lwd = linewidth,
       col = 3) # Plot u vs v
165 # mtext("u", side=1, line=3,las=1, col="black", cex = .75)
166 #
167 # means = apply(results, 1, function(x) mean(unlist(lapply(x, function(y) y[end.t])))) # Get
      means over many simulations
168 # stds = apply(results, 1, function(x) sd(unlist(lapply(x, function(y) y[end.t])))/sqrt(no.
      replicates)) # Ibid (stds)
169
170 ######## Ie = -1
171 # results = replicate(no.replicates, runSimulation(start.v, start.u, .1, -1))
172 # results.2 = replicate(no.replicates, runSimulation(start.v, start.u, .3, -1))
173 # results.3 = replicate(no.replicates, runSimulation(start.v, start.u, 1, -1))
174 # par(
175 #   mfrow = c(1, 1),
176 #   oma = c(10, 5, 4, 2) + 0.0,
177 #   mai = c(.0, .6, .0, .5)
178 # )
179 # linewidth = 2.5
```

```
# plot(seq(start.t,end.t - 1/print.ps, 1 / print.ps), results[,1]$u, type = "l", xaxt = "n", xlab = "n", ylab = "u", lwd = linewidth, col = 1) # Plot u
# plot(seq(start.t,end.t - 1/print.ps, 1 / print.ps), results[,1]$v, type = "l", xaxt = "n", xlab = "n", ylab = "v", lwd = linewidth, col = 2) # Plot v
# plot(results[,1]$u, results[,1]$v, type = "l", ylab = "v", xlab = "u", lwd = linewidth, col = 1, xaxt = "n") # Plot u vs v

# plot(results[,1]$u, results[,1]$v, type = "l", ylab = "v", xlab = "u", lwd = linewidth, col = 1, xlim = c(-2.1,0.5), ylim = c(-1.5,1.1), xaxt = "n") # Plot u vs v
# legend("topleft", c(expression(epsilon == 0.1), expression(epsilon == 0.3), expression(epsilon == 1.0)), col = c(1,2,3), inset = c(0.01, 0.05), lty = 1, lwd = linewidth, bg="white", cex = 1.3, box.lwd = 0)
# lines(results.2[,1]$u, results.2[,1]$v, type = "l", ylab = "v", xlab = "u", lwd = linewidth, xlim = c(-2.1,0.5), ylim = c(-1.5,1.1), col = 2, xaxt = "n") # Plot u vs v
# lines(results.3[,1]$u, results.3[,1]$v, type = "l", ylab = "v", xlab = "u", lwd = linewidth, xlim = c(-2.1,0.5), ylim = c(-1.5,1.1), col = 3) # Plot u vs v
# mtext("u", side=1, line=3,las=1, col="black", cex = .75)
```

oscillations.R

```
#!/usr/bin/env Rscript
setwd("~/compbio/src/assignments/cnsa2/code/")
library(RColorBrewer)
palette(brewer.pal(n = 8, name = "Set1"))
bias = TRUE
no.weights = 2 + ifelse(bias, 1, 0)
no.replicates = 100
lw.s = 2

classify = function(input, fct, ...) {
  ifelse(fct(input) < 0, -1, 1)
}

train.hebb = function(data, no.patterns, fct, ...) {
  targets = apply(data, 2, function(x) classify(x, fct=fct))
  weights = apply(data, 1, function(x) sum(targets * x))
  1/no.weights * weights
}

train.delta = function(data, no.patterns, learning.rate, no.iterations, fct, delta.fct, ...) {
  targets = apply(data, 2, function(x) classify(x, fct=fct)) # Find the correct answers
  weights = runif(no.weights, min=-0.01, max = 0.01) # Initialise weights
  for (ii in 1:no.iterations) {
    rand = sample(1:no.patterns, 1)
    output = classify(data[, rand] %*% weights, fct=fct)
    weights = weights + learning.rate * (targets[rand] - output) * delta.fct(data[, rand])
  }
  weights
}

train.delta.time = function(data, no.patterns, learning.rate, fct, delta.fct, ...) {
  targets = apply(data, 2, function(x) classify(x, fct=fct))
  weights = runif(no.weights, max = 0.01, min = -0.01)#0 # Initialise matrix
  outputs = apply(data, 2, function(x) classify(weights*x, fct=fct))
  iters = 1
  while (!all(targets == outputs)) {
    rand = sample(1:no.patterns, 1)
    weights = weights + learning.rate * (targets[rand] - classify(weights %*% delta.fct(data[,rand]), fct)) * delta.fct(data[, rand])
    outputs = apply(data, 2, function(x) classify(weights %*% x, fct=fct))
    # print(weights)
    iters = iters + 1
  }
  iters
}

train.perceptron = function(data, no.patterns, learning.rate, no.iterations, fct, ...) {
```

```
  targets = apply(data, 2, function(x) classify(x,  fct=fct))
  weights = runif(no.weights, min=-0.01, max = 0.01) # Initialise matrix
  for (ii in 1:no.iterations) {
    rand = sample(1:no.patterns, 1)
    output = classify(data[, rand] %*% weights, fct=fct)
    weights = weights + learning.rate / 2 * (targets[rand] - output) * data[, rand]
  }
  weights
}
train.perceptron.time = function(data, no.patterns, learning.rate, no.iterations, fct, ...) {
  targets = apply(data, 2, function(x) classify(x,  fct=fct))
  weights = runif(no.weights, min=-0.01, max = 0.01) # Initialise matrix
  outputs = apply(data, 2, function(x) classify(weights*x, fct=fct))
  iters = 1
  while (!all(targets == outputs)) {
    rand = sample(1:no.patterns, 1)
    weights = weights + learning.rate / 2 * (targets[rand] - outputs[rand]) * data[, rand]

    outputs = apply(data, 2, function(x) classify(weights %*% x, fct=fct))
    iters = iters + 1
  }
  iters
}

################################################################################
### Simulate for given number of patterns, with a specified
### training function and classification function.
################################################################################
run = function(no.patterns, train.fct, fct, learning.rate, delta.fct = function(x) x, ...) {
  # Generate data and set weights accordingly
  train.data = replicate(no.patterns, sample(c(1, -1), no.weights, replace = T))
  train.data[1,] = rep(1, no.patterns) #rbind(train.data, rep(1, no.patterns))
  test.data = replicate(no.patterns, sample(c(1, -1), no.weights, replace = T))
  test.data[1,] = rep(1, no.patterns) #rbind(test.data, rep(1, no.patterns))
  weights = train.fct(data=train.data, no.patterns=no.patterns, no.iterations = no.patterns*
      10, learning.rate=learning.rate, fct=fct, delta.fct=delta.fct)

  # Check how many patterns are recalled
  train.s = 0
  test.s = 0
  for (ii in 1:no.patterns) {
    if (classify(train.data[, ii] * weights, fct=fct) == classify(train.data[, ii], fct=fct))
      train.s = train.s + 1
    if (classify(test.data[, ii] * weights, fct=fct) == classify(test.data[, ii], fct=fct))
      test.s = test.s + 1
  }
  return(list(train = train.s, test = test.s))
}

run.time = function(no.patterns, train.fct, fct, learning.rate, delta.fct = function(x) x,
    ...) {
  # Generate data and set weights accordingly
  train.data = replicate(no.patterns, sample(c(1, -1), no.weights, replace = TRUE))
  iters = train.fct(data=train.data, no.patterns=no.patterns, learning.rate=learning.rate,
      fct=fct, delta.fct=delta.fct)
  iters
}

################################################################################
### RUN SIMULATIONS
################################################################################
patterns.stored = ceiling(2^(seq(1, 10, 1)))  # How many patterns stored?
pat.time = c(5, 10, 15, 20)
learning = 0.01
erf =  function(x) 2 * pnorm(x * sqrt(2)) - 1
delta.rule = function(x) 100*x #erf #1/(1 - exp(-x))

```

```
### SUM NORMAL
results.hebb = lapply(patterns.stored, function(x)
  lapply(1:no.replicates, function(y)
    run(
      no.patterns = x,
      train.fct = train.hebb,
      fct = sum,
      learning.rate = learning
    )))
results.delta = lapply(patterns.stored, function(x)
  lapply(1:no.replicates, function(y)
    run(
      no.patterns = x,
      train.fct = train.delta,
      fct = sum,
      learning.rate = learning,
      delta.fct = delta.rule # x # log(x + 10)
    )))
results.perceptron = lapply(patterns.stored, function(x)
  lapply(1:no.replicates, function(y)
    run(
      no.patterns = x,
      train.fct = train.perceptron,
      fct = sum,
      learning.rate = learning
    )))

### PROD NORMAL
results.prod.hebb = lapply(patterns.stored, function(x)
  lapply(1:no.replicates, function(y)
    run(
      no.patterns = x,
      train.fct = train.hebb,
      fct = prod,
      learning.rate = learning
    )))
results.prod.delta = lapply(patterns.stored, function(x)
  lapply(1:no.replicates, function(y)
    run(
      no.patterns = x,
      train.fct = train.delta,
      fct = prod,
      learning.rate = learning,
      delta.fct = delta.rule #function(x) log(x + 10)
    )))
results.prod.perceptron = lapply(patterns.stored, function(x)
  lapply(1:no.replicates, function(y)
    run(
      no.patterns = x,
      train.fct = train.perceptron,
      fct = prod,
      learning.rate = learning
    )))

### SUM TIME
time.results.delta = lapply(pat.time, function(x)
  lapply(1:no.replicates, function(y)
    run.time(
      no.patterns = x,
      train.fct = train.delta.time,
      fct = sum,
      learning.rate = learning,
      delta.fct = delta.rule #function(x) x #log(x + 10)
    )))
time.results.perceptron = lapply(pat.time, function(x)
  lapply(1:no.replicates, function(y)
    run.time(
```

```
178         no.patterns = x,
179         train.fct = train.perceptron.time,
180         fct = sum,
181         learning.rate = learning
182       )))
183
184 # ### PROD DELTA TIME
185 # time.results.prod.delta = lapply(pat.time, function(x)
186 #   lapply(1:no.replicates, function(y)
187 #     run.time(
188 #       no.patterns = x,
189 #       train.fct = train.delta.time,
190 #       fct = prod,
191 #       learning.rate = learning,
192 #       delta.fct = delta.rule #function(x) 1/(1 + exp(-x))
193 #     )))
194
195 # if(exists("../data/learning_raesults.RData")) {
196 #   load("../data/learning_results.RData")
197 # }
198
199 # save(results.hebb, results.delta, results.perceptron, time.results.delta, time.results.
      perceptron, results.prod.hebb, results.prod.delta, results.prod.perceptron,  file = "../
      data/learning_results.RData")
200 # time.results.prod.delta
201 ##############################################################################
202 ### CALCULATE MEANS + STDDEV
203 ##############################################################################
204 # Sum
205 hebb.train.means = sapply(results.hebb, function(x) mean(sapply(x, function(y) y$train))) /
      patterns.stored
206 hebb.train.stds = sapply(results.hebb, function(x) sd(sapply(x, function(y) y$train)) / sqrt(
      length(no.replicates))) / patterns.stored
207 hebb.test.means = sapply(results.hebb, function(x) mean(sapply(x, function(y) y$test)))/
      patterns.stored
208 hebb.test.stds = sapply(results.hebb, function(x) sd(sapply(x, function(y) y$test))/sqrt(
      length(no.replicates)))/patterns.stored
209 delta.train.means = sapply(results.delta, function(x) mean(sapply(x, function(y) y$train))) /
       patterns.stored
210 delta.train.stds = sapply(results.delta, function(x) sd(sapply(x, function(y) y$train)) /
      sqrt(length(no.replicates))) / patterns.stored
211 delta.test.means = sapply(results.delta, function(x) mean(sapply(x, function(y) y$test)))/
      patterns.stored
212 delta.test.stds = sapply(results.delta, function(x) sd(sapply(x, function(y) y$test))/sqrt(
      length(no.replicates)))/patterns.stored
213 perceptron.train.means = sapply(results.perceptron, function(x) mean(sapply(x, function(y) y$
      train))) / patterns.stored
214 perceptron.train.stds = sapply(results.perceptron, function(x) sd(sapply(x, function(y) y$
      train)) / sqrt(length(no.replicates))) / patterns.stored
215 perceptron.test.means = sapply(results.perceptron, function(x) mean(sapply(x, function(y) y$
      test)))/patterns.stored
216 perceptron.test.stds = sapply(results.perceptron, function(x) sd(sapply(x, function(y) y$test
      ))/sqrt(length(no.replicates)))/patterns.stored
217
218 # Sum time
219 delta.train.means.time = sapply(time.results.delta, function(x) mean(sapply(x, function(y) y)
      )) / pat.time
220 delta.train.stds.time = sapply(time.results.delta, function(x) sd(sapply(x, function(y) y)) /
       sqrt(length(no.replicates))) / pat.time
221 perceptron.train.means.time = sapply(time.results.perceptron, function(x) mean(sapply(x,
      function(y) y))) / pat.time
222 perceptron.train.stds.time = sapply(time.results.perceptron, function(x) sd(sapply(x,
      function(y) y)) / sqrt(length(no.replicates))) / pat.time
223
224 # Prod
225 hebb.train.prod.means = sapply(results.prod.hebb, function(x) mean(sapply(x, function(y) y$
      train))) / patterns.stored
```

```
hebb.train.prod.stds = sapply(results.prod.hebb, function(x) sd(sapply(x, function(y) y$train)) / sqrt(length(no.replicates))) / patterns.stored
hebb.test.prod.means = sapply(results.prod.hebb, function(x) mean(sapply(x, function(y) y$test)))/patterns.stored
hebb.test.prod.stds = sapply(results.prod.hebb, function(x) sd(sapply(x, function(y) y$test))/sqrt(length(no.replicates)))/patterns.stored
delta.train.prod.means = sapply(results.prod.delta, function(x) mean(sapply(x, function(y) y$train))) / patterns.stored
delta.train.prod.stds = sapply(results.prod.delta, function(x) sd(sapply(x, function(y) y$train)) / sqrt(length(no.replicates))) / patterns.stored
delta.test.prod.means = sapply(results.prod.delta, function(x) mean(sapply(x, function(y) y$test)))/patterns.stored
delta.test.prod.stds = sapply(results.prod.delta, function(x) sd(sapply(x, function(y) y$test))/sqrt(length(no.replicates)))/patterns.stored
perceptron.train.prod.means = sapply(results.prod.perceptron, function(x) mean(sapply(x, function(y) y$train))) / patterns.stored
perceptron.train.prod.stds = sapply(results.prod.perceptron, function(x) sd(sapply(x, function(y) y$train)) / sqrt(length(no.replicates))) / patterns.stored
perceptron.test.prod.means = sapply(results.prod.perceptron, function(x) mean(sapply(x, function(y) y$test)))/patterns.stored
perceptron.test.prod.stds = sapply(results.prod.perceptron, function(x) sd(sapply(x, function(y) y$test))/sqrt(length(no.replicates)))/patterns.stored

# Prod time
# delta.train.prod.means.time = sapply(time.results.prod.delta, function(x) mean(sapply(x, function(y) y))) / pat.time
# delta.train.prod.stds.time = sapply(time.results.prod.delta, function(x) sd(sapply(x, function(y) y)) / sqrt(length(no.replicates))) / pat.time

################################################################################
### PLOT
################################################################################
par(mfcol = c(3, 2), oma = c(10, 5, 4, 2) + 0.0, mai = c(.0, 0, .0, .0))
### SUM HEBB
plot(1, cex = 0, xlim = c(min(patterns.stored), max(patterns.stored)), ylim = c(0, 1), log = "x", xlab = "", ylab = "", bty = "n", xaxt = "n")
lines(patterns.stored, hebb.train.means, lwd = lw.s, col = 1)
suppressWarnings(arrows(patterns.stored, hebb.train.means - hebb.train.stds, patterns.stored, hebb.train.means + hebb.train.stds, length = 0.05, angle = 90, code = 3, cex = 2, lwd = lw.s, col = 1))
lines(patterns.stored, hebb.test.means, lwd = lw.s, col = 2)
suppressWarnings(arrows(patterns.stored, hebb.test.means - hebb.test.stds, patterns.stored, hebb.test.means + hebb.test.stds, length = 0.05, angle = 90, code = 3, cex = 2, lwd = lw.s, col = 2))
# legend("bottomright", c("Train", "Test"), col = 1:2, lty = 1, lwd = lw.s, inset = c(0.05, 0.05))

### SUM DELTA
plot(1, cex = 0, xlim = c(min(patterns.stored), max(patterns.stored)), ylim = c(0, 1), log = "x", xlab = "", ylab = "", bty = "n", xaxt = "n")
lines(patterns.stored, delta.train.means, lwd = lw.s, col = 1)
suppressWarnings(arrows(patterns.stored, delta.train.means - delta.train.stds, patterns.stored, delta.train.means + delta.train.stds, length = 0.05, angle = 90, code = 3, cex = 2, lwd = lw.s, col = 1))
lines(patterns.stored, delta.test.means, lwd = lw.s, col = 2)
suppressWarnings(arrows(patterns.stored, delta.test.means - delta.test.stds, patterns.stored, delta.test.means + delta.test.stds, length = 0.05, angle = 90, code = 3, cex = 2, lwd = lw.s, col = 2))
mtext("Fraction correct", side=2, line=3,las=3, col="black", cex = 1)
# legend("bottomright", c("Train", "Test"), col = 1:2, lty = 1, lwd = lw.s, inset = c(0.05, 0.05))

### SUM PERCEPTRON
plot(1, cex = 0, xlim = c(min(patterns.stored), max(patterns.stored)), ylim = c(0, 1), log = "x", xlab = "", ylab = "", bty = "n")
lines(patterns.stored, perceptron.train.means, lwd = lw.s, col = 1)
suppressWarnings(arrows(patterns.stored, perceptron.train.means - perceptron.train.stds,
```

```
    patterns.stored, perceptron.train.means + perceptron.train.stds, length = 0.05, angle =
    90, code = 3, cex = 2, lwd = lw.s, col = 1))
lines(patterns.stored, perceptron.test.means, lwd = lw.s, col = 2)
suppressWarnings(arrows(patterns.stored, perceptron.test.means - perceptron.test.stds,
    patterns.stored, perceptron.test.means + perceptron.test.stds, length = 0.05, angle = 90,
     code = 3, cex = 2, lwd = lw.s, col = 2))
# mtext("Patterns stored", side=1, line=3,las=1, col="black", cex = 1)
legend("bottomright", c("Train", "Test"), col = 1:2, lty = 1, lwd = lw.s, inset = c(0.05,
    0.05))

### PROD HEBB
plot(1, cex = 0, xlim = c(min(patterns.stored), max(patterns.stored)), ylim = c(0, 1), log =
    "x", xlab = "", ylab = "", bty = "n", xaxt = "n", yaxt = "n")
lines(patterns.stored, hebb.train.prod.means, lwd = lw.s, col = 1)
suppressWarnings(arrows(patterns.stored, hebb.train.prod.means - hebb.train.prod.stds,
    patterns.stored, hebb.train.prod.means + hebb.train.prod.stds, length = 0.05, angle = 90,
     code = 3, cex = 2, lwd = lw.s, col = 1))
lines(patterns.stored, hebb.test.prod.means, lwd = lw.s, col = 2)
suppressWarnings(arrows(patterns.stored, hebb.test.prod.means - hebb.test.prod.stds, patterns
    .stored, hebb.test.prod.means + hebb.test.prod.stds, length = 0.05, angle = 90, code = 3,
     cex = 2, lwd = lw.s, col = 2))
# legend("bottomright", c("Train", "Test"), col = 1:2, lty = 1, lwd = lw.s, inset = c(0.05,
    0.05))

### PROD DELTA
plot(1, cex = 0, xlim = c(min(patterns.stored), max(patterns.stored)), ylim = c(0, 1), log =
    "x", xlab = "", ylab = "", bty = "n", xaxt = "n", yaxt = "n")
lines(patterns.stored, delta.train.prod.means, lwd = lw.s, col = 1)
suppressWarnings(arrows(patterns.stored, delta.train.prod.means - delta.train.prod.stds,
    patterns.stored, delta.train.prod.means + delta.train.prod.stds, length = 0.05, angle =
    90, code = 3, cex = 2, lwd = lw.s, col = 1))
lines(patterns.stored, delta.test.prod.means, lwd = lw.s, col = 2)
suppressWarnings(arrows(patterns.stored, delta.test.prod.means - delta.test.prod.stds,
    patterns.stored, delta.test.prod.means + delta.test.prod.stds, length = 0.05, angle = 90,
     code = 3, cex = 2, lwd = lw.s, col = 2))
# mtext("Fraction correct", side=2, line=3,las=3, col="black", cex = 1)
# legend("bottomright", c("Train", "Test"), col = 1:2, lty = 1, lwd = lw.s, inset = c(0.05,
    0.05))

### PROD PERCEPTRON
plot(1, cex = 0, xlim = c(min(patterns.stored), max(patterns.stored)), ylim = c(0, 1), log =
    "x", xlab = "", ylab = "", bty = "n", yaxt = "n")
lines(patterns.stored, perceptron.train.prod.means, lwd = lw.s, col = 1)
suppressWarnings(arrows(patterns.stored, perceptron.train.prod.means - perceptron.train.prod.
    stds, patterns.stored, perceptron.train.prod.means + perceptron.train.prod.stds, length =
     0.05, angle = 90, code = 3, cex = 2, lwd = lw.s, col = 1))
lines(patterns.stored, perceptron.test.prod.means, lwd = lw.s, col = 2)
suppressWarnings(arrows(patterns.stored, perceptron.test.prod.means - perceptron.test.prod.
    stds, patterns.stored, perceptron.test.prod.means + perceptron.test.prod.stds, length =
    0.05, angle = 90, code = 3, cex = 2, lwd = lw.s, col = 2))
mtext("Patterns stored", side=1, line=3,las=1, col="black", cex = 1, at = 1.5)
legend("bottomright", c("Train", "Test"), col = 1:2, lty = 1, lwd = lw.s, inset = c(0.05,
    0.05))

################################################################################
### PLOT TIME
################################################################################
# par(mfrow = c(1, 1), oma = c(10, 5, 4, 2) + 0.0, mai = c(.0, .6, .0, .5))
# par(mfrow=c(1,1))
# # SUM DELTA
# plot(1, cex = 0, xlim = c(min(pat.time), max(pat.time)), ylim = c(0, 10), log = "", xlab =
    "", ylab = "", bty = "n")
# lines(pat.time, delta.train.means.time, lwd = lw.s, col = 1)
# suppressWarnings(arrows(pat.time, delta.train.means.time - delta.train.stds.time, pat.time,
     delta.train.means.time + delta.train.stds.time, length = 0.05, angle = 90, code = 3, cex
    = 2, lwd = lw.s, col = 1))
# mtext("Learning time", side=2, line=3,las=3, col="black", cex = 1)
```

```
#
# # SUM PERCEPTRON
# lines(pat.time, perceptron.train.means.time, lwd = lw.s, col = 2)
# suppressWarnings(arrows(pat.time, perceptron.train.means.time - perceptron.train.stds.time, pat.time, perceptron.train.means.time + perceptron.train.stds.time, length = 0.05, angle = 90, code = 3, cex = 2, lwd = lw.s, col = 2))

# # PROD DELTA
# lines(pat.time, delta.train.prod.means.time, lwd = lw.s, col = 3)
# suppressWarnings(arrows(pat.time, delta.train.prod.means.time - delta.train.prod.stds.time, pat.time, delta.train.prod.means.time + delta.train.prod.stds.time, length = 0.05, angle = 90, code = 3, cex = 2, lwd = lw.s, col = 3))
# mtext("Learning time", side=2, line=3,las=3, col="black", cex = 3)
# legend("bottomright", c("SDelta", "SPerceptron", "PDelta"), col = 1:3, lty = 1, lwd = lw.s, inset = c(0.05, 0.05))
```

perceptron_perceptron.R

```
#!/usr/bin/env Rscript
setwd("~/compbio/src/assignments/cnsa2/code/")
library(RColorBrewer)
palette(brewer.pal(n = 8, name = "Set1"))
learning.rate = 0.001
no.weights = 2
no.datapoints = 2000

# Get the Dayan/Abbott specified correlation matrix
correlation.matrix = function(data){
  t(data)%*%data / nrow(data)
}
covariance.matrix = function(data){
  new.vector = data - matrix(rep(colMeans(data),nrow(data)), byrow = T, ncol=2)
  t(new.vector)%*%(new.vector) / nrow(new.vector)
}

all.weights = matrix(NA, no.weights, no.datapoints + 1)
train = function(inputs, covorcor, method = "mult", ...) {
  weights = runif(no.weights, min = ifelse(method == "sub", 0, 1), max = 1)
  all.weights[,1] = weights
    init.weights = weights
   # weights = weights / sqrt(weights %*% weights)
  cor.mat = covorcor(inputs)
  if (method == "mult") {
    for (iter in 1:no.datapoints) {

        if(sum(weights) != 0)
          weights = weights + learning.rate * (c(cor.mat %*% weights) - sum(t(weights) %*% cor.mat) / sum(weights)  * weights)
        all.weights[,iter] = weights
      # print(sum(weights))
    }
  } else if (method == "sub") {
    for (iter in 1:no.datapoints) {
      weights = weights + learning.rate*(c(cor.mat%*%weights) - 1/no.weights * sum(cor.mat %*% weights))
      all.weights[,iter+1] = weights
      if(any(weights > 1 | weights < 0)){
        weights[weights > 1] = 1
        weights[weights < 0] = 0
        all.weights[,iter:no.datapoints] = weights
        break
      }
      all.weights[,iter + 1] = weights
    }
    }
  list(final=weights, init=init.weights, all.weights=all.weights)
}

```

```r
train2 = function(inputs, covorcor, method = "mult", no.iterations = 10000, weights, ...) {
  init.weights = weights
  all.weights = matrix(NA, no.weights, no.iterations + 1)
  all.weights[,1] = weights
  # weights = weights / sqrt(weights %*% weights)
  cor.mat = covorcor(inputs)
  if (method == "mult") {
    for (iter in 1:no.iterations) {
      # weights = weights + c(learning.rate * cor.mat %*% weights) - learning.rate * weights
          * c(weights %*% c(cor.mat %*% weights)) # eq. 11.30, works fine
      if(sum(weights) != 0)
        weights = weights + learning.rate * (c(cor.mat %*% weights) - sum(t(weights) %*% cor.
            mat) / sum(weights)  * weights)
      all.weights[,iter+1] = weights
    }
  } else if (method == "sub") {
    for (iter in 1:no.iterations) {
      weights = weights + learning.rate*(c(cor.mat%*%weights) - 1/no.weights * sum(cor.mat %*
          % weights))
      if(any(weights > 1 | weights < 0)){
        weights[weights > 1] = 1
        weights[weights < 0] = 0
        all.weights[,iter:no.datapoints] = weights
        break
      }
      all.weights[,iter+1] = weights
    }
  }
  list(final=weights / sqrt(weights[1]**2  + weights[2]**2), init=init.weights, all.weights =
       all.weights)
}

####################################
### SIMULATE
####################################
global.rho = .5
generate.data = function(rho, m){
  x = rnorm(no.datapoints, mean = m, sd = 1)
  y = rnorm(no.datapoints, mean = rho*x, sd = sqrt(1 - rho ** 2))
  data = cbind(x, y)
  data
}

m = 0
rho = global.rho
dat = generate.data(rho, m)
x = dat[,1]
y = dat[,2]

#############################
### Run simulation
#############################
par(mfrow = c(2, 2), oma = c(6, 5, 4, 2) + 0.0, mai = c(.0, .6, .2, .5))
plot(x, y, xlim = c(-3, 3) + mean(x), ylim = c(-4, 4) + mean(y), col = alpha(3, 1), cex = .5,
     bty ="n", xaxt = "n", ylab = "")
weights = train2(dat, correlation.matrix, method = "mult", weights = c(-1,-1))$final
# weights = weights / sqrt(weights[1]**2  + weights[2]**2)
lines(
   -3:3  + m,
   -3:3 * weights[2] / weights[1] + m*rho, #weights[2] + m * rho,
   col = 1,
  lwd = 5,
  lty = 1
)
weights = train(dat, covariance.matrix, method = "mult", weights = c(-1,-1))$final
# weights = weights / sqrt(weights %*% weights)
weights = weights / sqrt(weights[1]**2  + weights[2]**2)
```

```r
lines(
  -3:3 + m,
  -3:3 * weights[2] / weights[1] + m*rho, #weights[2] + m * rho,
  col = 2,
  lwd = 6,
  lty = 2
)

# Non-zero
m = 10
rho = global.rho
dat = generate.data(rho, m)
x = dat[,1]
y = dat[,2]

plot(x, y, xlim = c(-3, 3) + mean(x), ylim = c(-4, 4) + mean(y), col = 3, cex = .5, bty ="n",
     xaxt = "n", ylab = "")
weights = train(dat, correlation.matrix, method = "mult")$final
# weights = weights / sqrt(weights %*% weights)
weights = weights / sqrt(weights[1]**2 + weights[2]**2)

lines(
  -3:3 + m,
  -3:3 * weights[2] / weights[1] + mean(y), #weights[2] + m * rho,
  col = 1,
  lwd = 5,
  lty = 2
)

weights = train(dat, covariance.matrix, method = "mult")$final
# weights = weights / sqrt(weights %*% weights)
weights = weights / sqrt(weights[1]**2 + weights[2]**2)
lines(
  -3:3 + m,
  -3:3 * weights[2] / weights[1] + mean(y), #weights[2] + m * rho,
  col = 2,
  lwd = 5,
  lty = 3, ylab = ""
)
# legend("bottomright", c("Cor", "Cov"), lty = c(1,2), lwd = c(6,5), col = 1:2, inset = c
    (0.02,0.02))


m = 0
rho = global.rho
dat = generate.data(rho, m)
x = dat[,1]
y = dat[,2]

##############################
### Run simulation
##############################
plot(x, y, xlim = c(-3, 3) + mean(x), ylim = c(-4, 4) + mean(y), col = alpha(3, 1), cex = .5,
     bty ="n", ylab = "")
weights = train(dat, correlation.matrix, method = "sub")$final
# weights = weights / sqrt(weights[1]**2 + weights[2]**2)
print(weights)

lines(
  -3:3 + m,
  -3:3 * weights[2] / ifelse(weights[1]==0, 0.00000001, weights[1]) + m*rho, #weights[2] + m
      * rho,
  col = 1,
  lwd = 5,
  lty = 2
)
weights = train(dat, covariance.matrix, method = "sub")$final
```

```
# weights = weights / sqrt(weights %*% weights)
# weights = weights / sqrt(weights[1]**2  + weights[2]**2)
lines(
  -3:3  + m,
  -3:3 * weights[2] / ifelse(weights[1]==0, 0.00000001, weights[1]) + m*rho, #weights[2] + m * rho,
  col = 2,
  lwd = 6,
  lty = 3
)
mtext(side = 2, line = 3, at = 5, las = 3, "y")
mtext(side = 1, line = 3, at = 5, las = 1, "x")
mtext(side = 2, line = 4, at = 10, las = 3, "multiplicative")
mtext(side = 2, line = 4, at = 0, las = 3, "subtractive")
# print(weights)



# Non-zeor
m = 10
rho = global.rho
dat = generate.data(rho, m)
x = dat[,1]
y = dat[,2]

plot(x, y, xlim = c(-3, 3) + mean(x), ylim = c(-4, 4) + mean(y), col = 3, cex = .5, bty ="n", ylab = "")
weights = train(dat, correlation.matrix, method = "sub")$final
# weights = weights / sqrt(weights %*% weights)
# weights = weights / sqrt(weights[1]**2  + weights[2]**2)
print(weights)

lines(
  -3:3  + m,
  -3:3 * weights[2] / ifelse(weights[1]==0, 0.00000001, weights[1]) + mean(y), #weights[2] + m * rho,
  col = 1,
  lwd = 5,
  lty = 2
)

weights = train(dat, covariance.matrix, method = "sub")$final
# weights = weights / sqrt(weights %*% weights)
# weights = weights / sqrt(weights[1]**2  + weights[2]**2)
lines(
  -3:3  + m,
  -3:3 * weights[2] / ifelse(weights[1]==0, 0.00000001, weights[1]) + mean(y), #weights[2] + m * rho,
  col = 2,
  lwd = 5,
  lty = 3
)
print(weights)
legend("bottomright", c("Cor", "Cov"), lty = c(1,2), lwd = c(6,5), col = 1:2, inset = c(0.02,0.02))

print(prcomp(data.frame(dat)))



##################
par(mfrow=c(2,2))
w.seq = seq(-1,1, 0.2)
dat = generate.data(.5, 10)
w.mat = sapply(w.seq, function(x) sapply(w.seq, function(y) prod(train2(dat, correlation.matrix, method = "mult", weights =c(x,y))$final)))
image(w.mat, xaxt = "n", yaxt = "n", col = heat.colors(12))
```

```
mtext(side=3, line = 1, "correlation", las = 1)
mtext(side=2, line = 3, "multiplicative", las = 3)
axis(2, at=seq(0,1,.25), labels = seq(-1,1, 0.5))
w.mat = sapply(w.seq, function(x) sapply(w.seq, function(y) prod(train2(dat, covariance.
    matrix, method = "mult", weights =c(x,y))$final)))
image(w.mat, xaxt = "n", yaxt = "n", col = heat.colors(12))
mtext(side=3, line = 1, "covariance", las = 1)

w.seq = seq(0,1, 0.1)
w.mat = sapply(w.seq, function(x) sapply(w.seq, function(y) train2(dat, correlation.matrix,
    method = "sub", weights =c(x,y))$final[1]))
image(w.mat, xaxt = "n", yaxt = "n", col = heat.colors(12))
axis(2, at=seq(0,1,.25), labels = seq(0,1, 0.25))
axis(1, at=seq(0,1,.25), labels = seq(-1,1, 0.5))
mtext(side=2, line = 3, "subtractive", las = 3)
w.mat = sapply(w.seq, function(x) sapply(w.seq, function(y) train2(dat, covariance.matrix,
    method = "sub", weights =c(x,y))$final[1]))
image(w.mat, xaxt = "n", yaxt = "n", col = heat.colors(12))
axis(1, at=seq(0,1,.25), labels = seq(-1,1, 0.5))
#

lw.s = 3
par(mfrow = c(2, 2), oma = c(6, 5, 2, 4) + 0.0, mai = c(.2, .2, .2, .2))
xmax = 30
w.weights = train2(dat, correlation.matrix, method = "mult", weights =c(.9, .1))$all.weights
plot(w.weights[1,], type = "l", ylim = c(-1, 1), col = 1, xlim = c(0, xmax), lwd = lw.s, xaxt
    = "n")
lines(w.weights[2,], col = 1, lwd = lw.s)
mtext(side = 2, at = .0, "multiplicative", las = 3, line = 4)
w.weights = train2(dat, correlation.matrix, method = "mult", weights =c(.9, .1))$all.weights
lines(w.weights[1,], type = "l", ylim = c(-1, 1), col = 2, xlim = c(0, xmax), lwd = lw.s,
    xaxt = "n", yaxt = "n", ylab = "")
lines(w.weights[2,], col = 2, lwd = lw.s)

w.weights = train2(dat, correlation.matrix, method = "mult", weights =c(-.6, .5))$all.weights
plot(w.weights[1,], type = "l", ylim = c(-1, 1), col = 1, xlim = c(0, xmax), lwd = lw.s, xaxt
    = "n", yaxt = "n", ylab = "")
lines(w.weights[2,], col = 1, lwd = lw.s)
w.weights = train2(dat, correlation.matrix, method = "mult", weights =c(-.5, .6))$all.weights
lines(w.weights[1,], type = "l", ylim = c(-1, 1), col = 2, xlim = c(0, xmax), lwd = lw.s,
    xaxt = "n", yaxt = "n", ylab = "")
lines(w.weights[2,], col = 2, lwd = lw.s)

w.weights = train2(dat, correlation.matrix, method = "sub", weights =c(.1,.9))$all.weights
plot(w.weights[1,], type = "l", ylim = c(0, 1), col = 1, xlim = c(0, xmax), lwd = lw.s)
lines(w.weights[2,], col = 1, lwd = lw.s)
mtext(side = 2, at = 1, "Weight", las = 3, line = 3)
mtext(side = 2, at = .5, "subtractive", las = 3, line = 4)

w.weights = train2(dat, correlation.matrix, method = "sub", weights =c(.4,.3))$all.weights
plot(w.weights[1,], type = "l", ylim = c(0, 1), col = 1, xlim = c(0, xmax), lwd = lw.s, yaxt
    = "n", ylab = "")
lines(w.weights[2,], col = 1, lwd = lw.s)
mtext(side = 1, at = -1.5, "Iteration", las = 1, line = 3)



# image(w.mat, xaxt = "n", yaxt = "n", col = 1:8)
# mtext(side=3, line = 1, "correlation", las = 1)
# mtext(side=2, line = 3, "multiplicative", las = 3)
# axis(2, at=seq(0,1,.25), labels = seq(-1,1, 0.5))
# w.mat = sapply(w.seq, function(x) sapply(w.seq, function(y) sum(train2(dat, covariance.
    matrix, method = "mult", weights =c(x,y))$final)))
# image(w.mat, xaxt = "n", yaxt = "n", col = 1:8)
# mtext(side=3, line = 1, "covariance", las = 1)

```

```
# image(w.mat, xaxt = "n", yaxt = "n", col = 1:8)
# axis(2, at=seq(0,1,.25), labels = seq(0,1, 0.25))
# axis(1, at=seq(0,1,.25), labels = seq(-1,1, 0.5))
# mtext(side=2, line = 3, "subtractive", las = 3)
# w.mat = sapply(w.seq, function(x) sapply(w.seq, function(y) train2(dat, covariance.matrix,
    method = "sub", weights =c(x,y))$final[1]))
# image(w.mat, xaxt = "n", yaxt = "n", col = 1:8)
# axis(1, at=seq(0,1,.25), labels = seq(-1,1, 0.5))
```

unsup_mult.R

```
#!/usr/bin/env Rscript
setwd("~/compbio/src/assignments/cnsa2/code/")
library(RColorBrewer)
palette(brewer.pal(n = 8, name = "Set1"))
par(mfrow = c(1, 1), oma = c(6, 5, 4, 2) + 0.0, mai = c(.0, .6, .0, .5))

# Parameters
no.datapoints = 2000
no.iter.ocdom = 1000
no.cells = 512
rho =  .5
m = 0
sigma = 0.066
learning.rate = 0.01

# Circumference and inter-neuronal distances
a = 10/(no.cells)
total.circ = 10 #+ a

# Preallocate
wminus = matrix(NA, no.cells, no.iter.ocdom)
K = matrix(0, no.cells, no.cells)
K = sapply(1:no.cells, function(ii)
  sapply(1:no.cells, function(jj)
    exp(
      -ifelse(
        abs(ii - jj) < no.cells / 2,
        a * abs(ii - jj),
        total.circ - a * abs(ii - jj)
      ) ** 2 / (2 * sigma ** 2)
    ) - 1 / 9 * exp(
      -ifelse(
        abs(ii - jj) < no.cells / 2,
        a * abs(ii - jj),
        total.circ - a * abs(ii - jj)
      ) ** 2 / (18 * sigma ** 2)
    )))

# Train the weights!
train = function(weights = matrix(runif(no.cells * 2, min = 0, max = 0.01), no.cells, 2),
                 data = cbind(rnorm(no.datapoints, mean = m, sd = 1),
                              rnorm(no.datapoints, mean = rho * x, sd = sqrt(1 - rho ** 2))),
                 ...) {
  Q = cor(data)

  # Update weights
  for (iter in 1:no.iter.ocdom) {
    change = learning.rate * K%*%weights%*%Q # Just calculate this once
    weights = weights +  change - rowMeans(change)
    weights[(weights > 1)] = 1
    weights[(weights < 0)] = 0
    wminus[, iter] = weights[, 2] - weights[, 1]
  }
  list(weights=weights, wminus=wminus)
}

no.replicates = 100
```

```
# results = replicate(no.replicates, train())
# save(results, file="../data/ocdom_results.RData")
# load("../data/ocdom_results.RData")

# Perform fourier transform on the 100 iterations.
last.wminus = sapply((results["wminus",]), function(x) x[, no.iter.ocdom]) # Get the final wminus
ffts = apply(last.wminus, 2, fft)
ffts = apply(ffts, 1:2, function(x) Re(sqrt(x**2)))
means = apply(ffts, 1, mean)
stds = apply(ffts, 1, function(x) sd(x) / sqrt(no.replicates))

lw.s = 2
par(mfrow=c(2,1))
plot(means, type = "l", col = 1, lwd = lw.s, xaxt = "n")#, ylim = c(-.1,1))
plot(K[,1], col = 2, lwd = lw.s, type = "l", xlab = "Neuron index", ylab = "Magnitude")
lines(K[,128], col = 3, lwd = lw.s, type = "l", xlab = "Neuron index", ylab = "Magnitude")
lines(K[,256], col = 4, lwd = lw.s, type = "l", xlab = "Neuron index", ylab = "Magnitude")
lines(K[,384], col = 5, lwd = lw.s, type = "l", xlab = "Neuron index", ylab = "Magnitude")
legend("bottomright", col=1:5, bg = "white", lwd = lw.s, c(expression("fft(w"["-"]*")"), expression("K"[1]), expression("K"[128]),expression("K"[256]),expression("K"[384])), lty = c(1), inset = c(0.02,0.02))
# suppressWarnings(arrows(1:no.iter.ocdom, means - stds, 1:no.iter.ocdom, means + stds, length = 0.05, angle = 90, code = 3, cex = 2, lwd = lw.s, col = 1))
mtext(side=1, las = 1, line = 3, "Iteration")

##############################
### Run simulation and plot
##############################
par(mfrow = c(2, 1), oma = c(6, 5, 0, 2) + 0.0, mai = c(.0, .2, .3, .4))
plot(1, xlim=c(1, no.iter.ocdom), ylim =c(-1,1), cex = 0, bty = "n", xaxt = "n")
mtext(side=2, las = 3, line = 3, "Weight")
train.out = train()
weights = train.out[[1]]
wminus = train.out[[2]]
tmp = sapply(1:no.cells, function(cell) lines(1:no.iter.ocdom, wminus[cell, ], cex = 0.1, col = sample(1:10)))
image(t(wminus), ylab = "", yaxt="n", xlab = "Iterations", xaxt = "n", bty= "n")
mtext(side=2, las = 3, line = 3, "Index")
axis(1, at=seq(0, 1,.20), labels = seq(0, no.iter.ocdom, no.iter.ocdom/5))
axis(2, at=seq(0, 1,.25), labels=c(1, seq(512/4, 512, 512/4)))
mtext(side=1, las = 1, line = 3, "Iteration")



### Save .GIF
# setwd("~/gif")
# frames = 512
# for(i in 1:frames){
#   # creating a name for each plot file with leading zeros
#   if (i < 10) {
#     name = paste('000', i, 'plot.png', sep = '')
#   }
#   else if (i < 100 && i >= 10) {
#     name = paste('00', i, 'plot.png', sep = '')
#   }
#   else if (i >= 100) {
#     name = paste('0', i, 'plot.png', sep = '')
#   }
#
#   #saves the plot as a .png file in the working directory
#   png(name)
#   plot(fft(K[i,]), type = "l")
#   dev.off()
# }
```

```
120 # cat("sub+cov unnorm:\t", weights, "\n")
121 # weights = weights / sqrt(weights %*% weights)
122 # cat("sub+cov norm:\t", weights, "\n")
123 # print(any(weights != 1))
124 # lines(-3:3 * weights[1] + m,-3:3 * weights[2] + m * rho, col = 1, lwd = 5, lty = 3)
```

ocdom.R