Submission information

Course	EE538 Neural Networks
Section	
Submission title	Homework 2
Date of submission	March 26, 2021

Student information

Name	**ID**
Quang Minh Nguyen	20200854

Homework 2

Quang Minh Nguyen

March 26, 2021

1

(a) The plot of generated data and computed eigenvectors is included in Figure 2. Figure 1 shows numerical values of the eigenvectors.

For the code, see `P1.py`. For a larger figure, see `P1a.jpg`.

```
[[ 7.12925670e-01 -7.01239607e-01 -4.49152803e-05]
 [ 7.01200987e-01  7.12887075e-01 -1.04495693e-02]
 [ 7.35967142e-03  7.41827158e-03  9.99945401e-01]]
```

Figure 1: Problem 1a—Computed eigenvectors. Each line is a vector.

(b) Our goal is to derive the learning rule

$$w_i(n+1) = w_i(n) + \mu y(n) x_i'(n)\,, \tag{1}$$

where $x_i'(n) = x_i(n) - y(n)w_i(n)$.

Let's start by describing the linear model we will use,

$$y = \mathbf{w}^\top \mathbf{x}\,,$$

where $\mathbf{w}, \mathbf{x} \in \mathbb{R}^m$.

We want to adapt the weight w according to the Hebbian learning rule, i.e.,

$$w_i(n+1) = w_i + \mu(n) y(n) x_i(n)\,.$$

However, this rule may lead to uncontrolled growth of the weight. We prevent this by renormalizing w at each time step. The modified learning rule is then

$$w_i(n+1) = \frac{w_i + \mu(n) y(n) x_i(n)}{(\sum_{i=1}^m (w_i + \mu(n) y(n) x_i(n))^2)^{1/2}}\,. \tag{2}$$

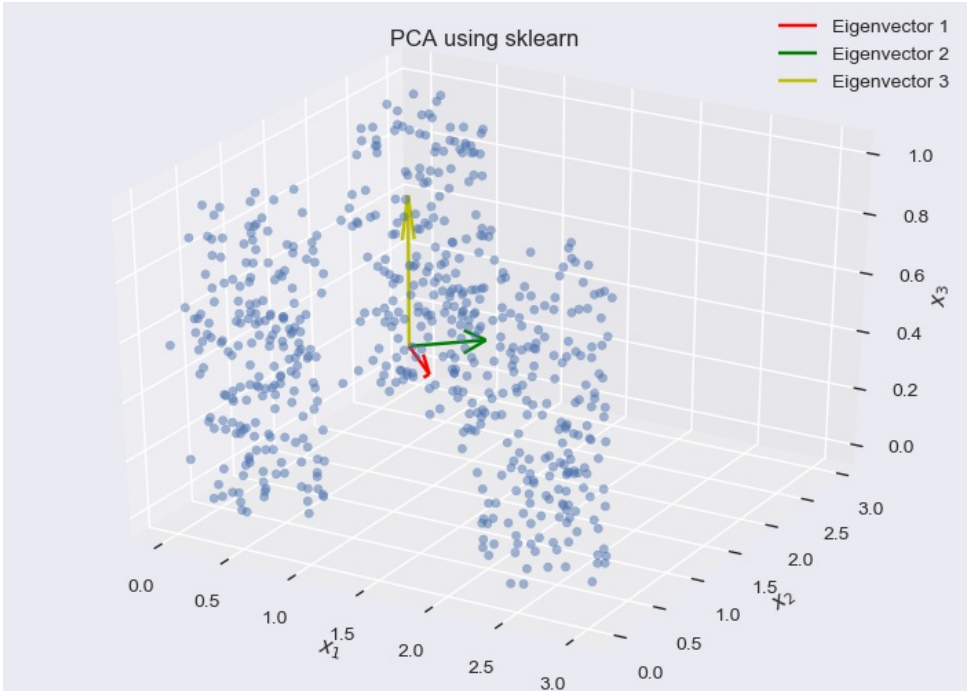

Figure 2: Problem 1a—Generated data and computed eigenvectors using open-source library `scikit-learn`

This normalization may slow down our computation. We will find a simpler approximation. Consider when $\mu(n)$ is small enough for us to ignore any second-order term of it. We now rewrite the denominator:

$$(\sum_{i=1}^{m}(w_i+\mu(n)y(n)x_i(n))^2)^{1/2} = (\sum_{i=1}^{m} w_i^2+2\mu(n)y(n)\sum_{i=1}^{m} w_i(n)x_i(n)+\mu^2(n)y^2(n)\sum_{i=1}^{m} w_i x_i(n))^{1/2}\,.$$

The last term inside the $(.)^{1/2}$ is proportional to $\mu(n)^2$ and therefore can be ignored. The sum $\sum_i^m w_i x_i(n)$ in the second term also has a very concise representation: $y(n)$. Furthermore, as we renormalize at each iteration, $\sum_i^m w_i^2$ is nothing other than 1. All of these allow us to write

$$(\sum_{i=1}^{m}(w_i + \mu(n)y(n)x_i(n))^2)^{1/2} = (1 + 2\mu(n)y^2(n))^{1/2}\,.$$

Since $\mu(n)$ is a very small number, so is $2\mu(n)y^2(n)$. A simple binomial approximation tells us that

$$(\sum_{i=1}^{m}(w_i + \mu(n)y(n)x_i(n))^2)^{1/2} = (1 + 2\mu(n)y^2(n))^{1/2} = 1 + \mu(n)y^2(n).$$

Before going back to the learning rule, let us make a final approximation:

$$1 + \mu(n)y^2(n) = \frac{1-\mu^2(n)y^4(n)}{1-\mu(n)y^2(n)} = \frac{1}{1-\mu(n)y^2(n)}\,.$$

Plugging this back into (2), we get

$$\begin{aligned} w_i(n+1) &= (w_i + \mu(n)y(n)x_i(n))(1 - \mu(n)y^2(n)) \\ &= w_i + \mu(n)y(n)x_i(n) - w_i\mu(n)y^2(n) - \mu^2(n)y^3(n)x_i(n) \\ &= w_i + \mu(n)y(n)(x_i(n) - y(n)w_i(n))\,, \end{aligned}$$

where the third equality, yet again, is an approximation. This is exactly the learning rule we desire as in (1).

(c) Since the algorithm of part (b) is a special case of what we describe in part (d), the results are combined with part (e). See part (e) below.

(d) A naive strategy would be to (1) apply the learning rule of part (b) till convergence at a weight $\mathbf{w}_1 \in \mathbb{R}^m$, (2) for each input $\mathbf{x} \in \mathbb{R}^m$, project it into the orthogonal complement of $\text{span}\{\mathbf{w}_1\}$, and (3) proceed with the same algorithm, again. In formula, we write

$$\Delta\mathbf{w}_2(n) = \mu(n)y_2(n)[(\mathbf{x}(n) - \mathbf{w}_1^\top\mathbf{x}(n)\mathbf{w}_1) - y_2(n)\mathbf{w}_2(n)]\,, \tag{3}$$

where

$$y_2 = \mathbf{w}_2^\top\mathbf{x}$$

is our model to learn the second principal component. Note that we now denote the model of part (b) in a similar fashion: $y_1 = \mathbf{w}_1^\top\mathbf{x}$.

However, another possible strategy would be to learn the two principal components *simultaneously*. That is, to apply (3) as well as (1) at each time step. After all, the learning rule at (1) works independently with second principal component's counterpart. Since the former is guaranteed to converge, learning rule 3 should then, at the latest, start to converge in a situation identical to the naive strategy we described.

So to summarize, we initialize weights $\mathbf{w}_1, \mathbf{w_2} \in \mathbb{R}^m$, then at each time step update

$$\begin{cases} \Delta\mathbf{w}_1(n) &= \mu(n)y_1(n)(\mathbf{x}(n) - y_1(n)\mathbf{w}_1(n))\,, \\ \Delta\mathbf{w}_2(n) &= \mu(n)y_2(n)(\mathbf{x}(n) - y_1(n)\mathbf{w}_1(n) - y_2(n)\mathbf{w}_2(n))\,. \end{cases}$$

till convergence.

(e) The two eigenvectors learned using generalized Hebbian algorithm, i.e., part (b) and (d), are plotted in Figure 4. We also plot the learning curve, i.e., distance between actual eigenvectors and learned eigenvectors, in Figure 5. The algorithm converges quickly after the first few epochs (each epoch is 600 iterations). The numerical results are shown in Figure 3.

For the code, see `P1.py`. For larger figures, see `P1ce.jpg` and `P1ceE.jpg`.

2

(a) We will find the solution to the differential equation

$$\tau_m\frac{\mathrm{d}u}{\mathrm{d}t} = -u + V_i\,. \tag{4}$$

```
[[ 0.71332881 -0.70091227  0.00242193]
 [ 0.70108256  0.71325249 -0.01363302]]
```

Figure 3: Problem 1c and 1e—Two learned principal components. Each line is a vector.

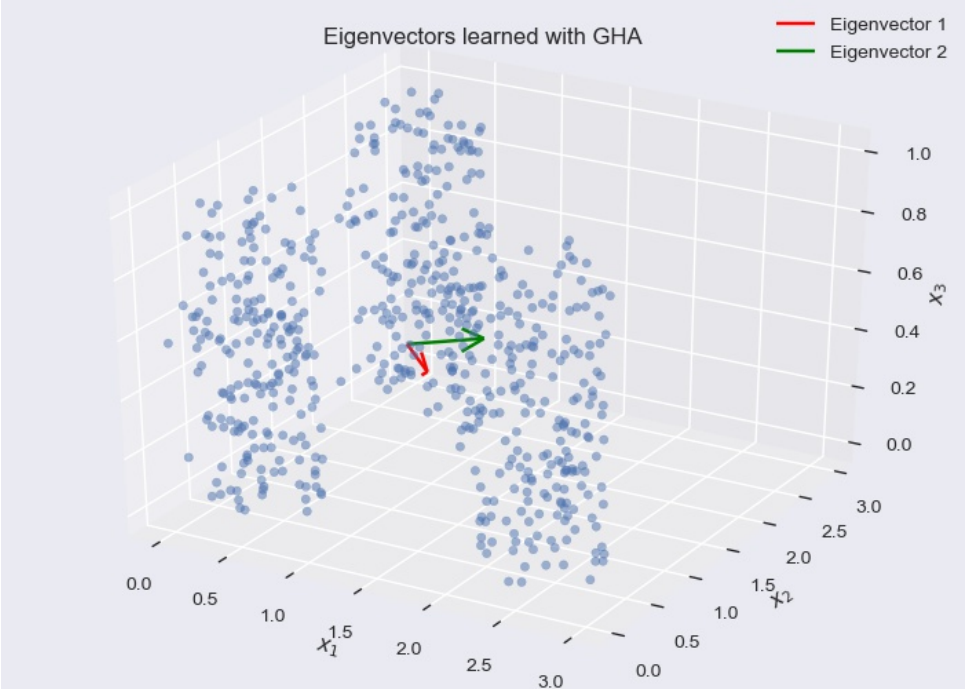

Figure 4: Problem 1c and 1e—Learned eigenvectors using generalized Hebbian algorithm

Multiplying both sides of (4) by $e^{t/\tau_m}/\tau_m$ yields

$$\begin{aligned} e^{t/\tau_m}\frac{\mathrm{d}u}{\mathrm{d}t} + u\frac{\mathrm{d}e^{t/\tau_m}}{\mathrm{d}t} &= \frac{V_i}{\tau_m}e^{t/\tau_m} \\ \Leftrightarrow \frac{\mathrm{d}(ue^{t/\tau_m})}{\mathrm{d}t} &= \frac{V_i}{\tau_m}e^{t/\tau_m} \\ \Leftrightarrow ue^{t/\tau_m} &= V_i e^{t/\tau_m} + c \\ \Leftrightarrow u &= V_i + \frac{c}{e^{t/\tau_m}}\,. \end{aligned}$$

Combined with the initial condition $u(0) = 0$, this gives us

$$u = V_i - \frac{V_i}{e^{t/\tau_m}}\,,$$

which is the formula of u we look for.

(b) The plot is included in Figure 6. For the code, see `P2b.py`. For a larger figure, see `P2b.jpg`.

(c) Note that when $V_i \leq \theta$, u will never reach the threshold and hence the neuron never fires. For now, assume that $V_i > \theta$. The time T it takes for the membrane potential to reach the threshold θ is determined from the equation

$$\theta = V_i - \frac{V_i}{e^{T/\tau_m}}\,. \tag{5}$$

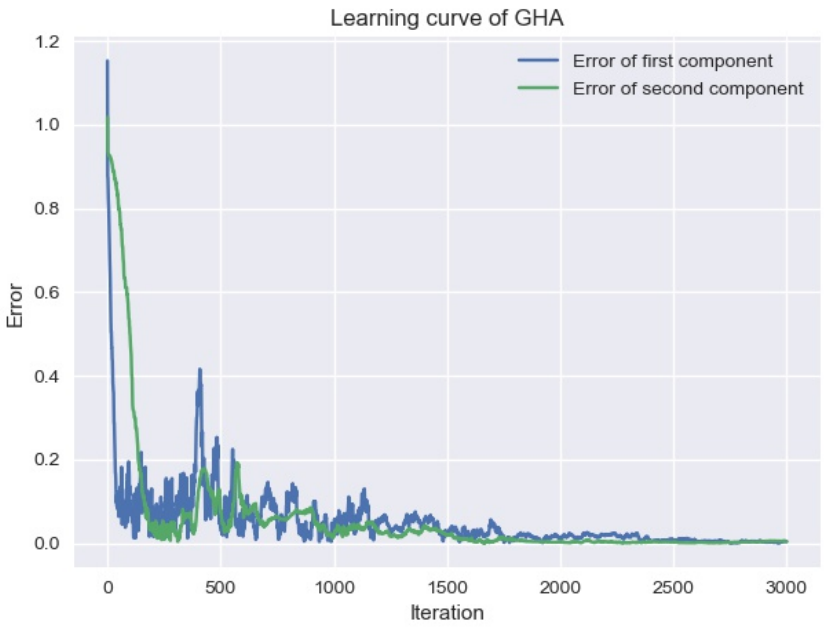

Figure 5: Problem 1c and 1e—Error curve of generalized Hebbian algorithm

Rearranging terms in (5) yields

$$e^{T/\tau_m} = \frac{V_i}{V_i - \theta}\,.$$

In other words,

$$T = \tau_m \ln \frac{V_i}{V_i - \theta}\,.$$

So the number of spikes per unit time, i.e., the firing frequency, is

$$f(V_i) = \begin{cases} \left(\tau_m \ln \frac{V_i}{V_i - \theta}\right)^{-1} & ,\ V_i > \theta\,, \\ 0 & ,\ \text{otherwise}\,. \end{cases}$$

(d) The plot is included in Figure 7. At large values of V_i, the linear behavior comes from

$$f(V_i) \approx \frac{V_i - \theta}{\tau_m \theta}\,.$$

We can see a resemblance of the Rectified Linear Unit.

For the code, see `P2d.py`. For a larger figure, see `P2d.jpg`.

(e) The plot is included in Figure 8.

For the code, see `P2e.py`. For a larger figure, see `P2e.jpg`.

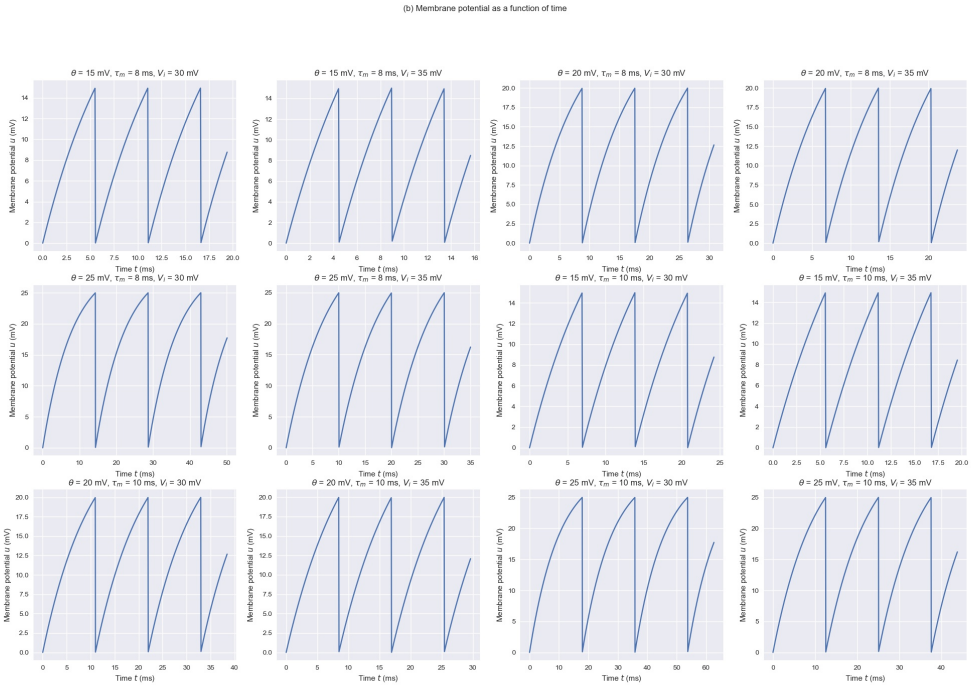

Figure 6: Problem 2b—Membrane potential plotted against time

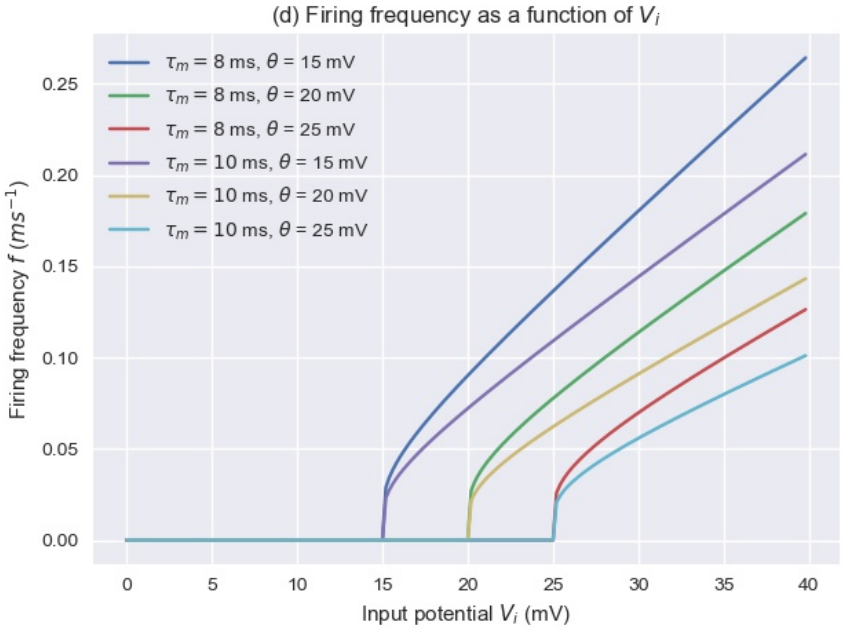

Figure 7: Problem 2d—Number of spikes per unit time plotted against input potential

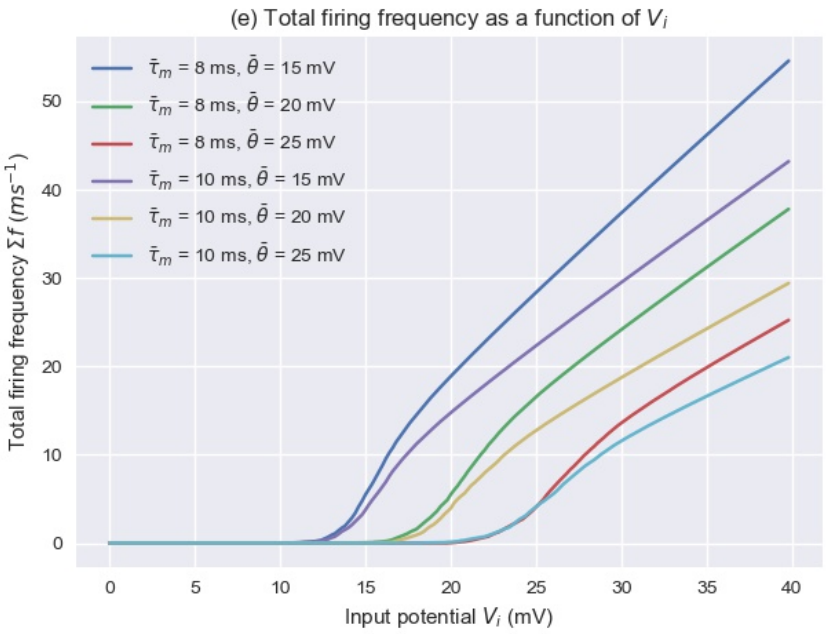

Figure 8: Problem 2e—Total number of spikes per unit time plotted against input potential