Machine Learning
Homework #4

Daniel Ho Kwan Leung
104360098
CSIE, Year 3

Question 1:

The learning rate of SOFM networks is,

$$learning\ rate = \eta(t)\gamma_{(i,j)}(t)$$

$$\eta(t) = \eta_0 e^{\frac{t}{\lambda}}$$

$$\gamma_{(i,j)}(t) = e^{\left[\frac{-\sqrt{(i-i_0)^2+(j-j_0)^2}}{2[\sigma(t)]^2}\right]}$$

$$\sigma(t) = \sigma_0 e^{-\frac{t}{\lambda}}$$

According to the question, $\eta_0 = \sigma_0 = 0.1, \lambda = 10$. The BMU is located at $(x_0 \pm 1, y_0)\ or\ (x_0, y_0 \pm 1)$ where (x_0, y_0) is the coordinate of the BMU. Therefore,

$$\eta(t) = 0.1e^{\frac{t}{10}}$$

$$\sigma(t) = 0.1e^{-\frac{t}{10}}$$

$$\gamma_{(i,j)}(t) = e^{\left[\frac{-\sqrt{(i-i_0)^2+(j-j_0)^2}}{2[\sigma(t)]^2}\right]}$$

$$= e^{\left[-\frac{1}{2\left[0.1e^{-\frac{t}{10}}\right]^2}\right]}$$

$$= e^{\left[-\frac{1}{2(0.01)e^{-\frac{t}{5}}}\right]}$$

$$= e^{\left[-50e^{\frac{t}{5}}\right]}$$

$$learning\ rate = 0.1e^{\frac{t}{10}}e^{\left[-50e^{\frac{t}{5}}\right]}$$

We have to find the required iteration such that the learning rate of the node next to BMU is less than 0.001.

$$0.1e^{\frac{t}{10}}e^{\left[-50e^{\frac{t}{5}}\right]} < 0.001$$

Since it can't be easily solved. By using program, the code was uploaded to GitHub:

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW4_Q1.ipynb

```
import numpy as np
```

```
def learning_rate(t):
    eta_0 = 0.1
    sigma_0 = 0.1
    lamb = 10.
    eta = eta_0 * np.exp(t / lamb)
    sigma = sigma_0 * np.exp(-(t / lamb))
    gamma = np.exp(-(1 / (2 * sigma**2)))
    lr = eta * gamma
    return lr
```

```
for i in range(1, 11):
    print('t = {}, learning rate = {}'.format(i, learning_rate(i)))
```

The outputs are,

```
t = 1, learning rate = 3.318986956663983e-28
t = 2, learning rate = 4.9237635474066944e-34
t = 3, learning rate = 3.6600020233331726e-41
t = 4, learning rate = 7.026039192832359e-50
t = 5, learning rate = 1.550269929287323e-60
t = 6, learning rate = 1.462694903717886e-73
t = 7, learning rate = 1.7638339865845787e-89
t = 8, learning rate = 6.218749388550233e-109
t = 9, learning rate = 1.057887464553933e-132
t = 10, learning rate = 9.615679289785645e-162
```

Therefore, the required iteration is 1 since when t = 1, the learning rate is already less than 0.001.

Question 2:

Before proving $\alpha_w = \frac{1}{m}\sum_{i=1}^{m}\beta_{i,w}$, we have to define the notation and tidy up the information known.

Assume we have a dataset denoted by $X_{m\times n}$ which has m tuples of datum and n features. Let the number of clusters be c.

$$Gaussian\ Distribution = \mathcal{N}_j\left(x^{(i)}|\mu_j,\Sigma_j\right) = \frac{1}{\sqrt{(2\pi)^n|\Sigma|}}e^{\left[-\frac{1}{2}\left(x^{(i)}-\mu_j\right)^T\Sigma_j^{-1}\left(x^{(i)}-\mu_j\right)\right]}$$

$$\theta = (\mu_1,\dots,\mu_c,\Sigma_1,\dots,\Sigma_c,\alpha_1,\dots,\alpha_c)$$

$$Probability\ Density\ Function\ of\ Gaussian\ Mixture\ Model = p\left(x^{(i)}|\theta\right) = \sum_{j=1}^{c}\alpha_j\mathcal{N}_j\left(x^{(i)}|\mu_j,\Sigma_j\right)$$

$$\beta_{i,j} = \frac{\alpha_j\mathcal{N}_j\left(x^{(i)}|\mu_j,\Sigma_j\right)}{\sum_{k=1}^{c}\alpha_k\mathcal{N}_k(x^{(i)}|\mu_k,\Sigma_k)}$$

$$\mu_j = \frac{\sum_{i=1}^{m}\beta_{i,j}x^{(i)}}{\sum_{i=1}^{m}\beta_{i,j}}$$

$$\Sigma_j = \frac{1}{\sum_{i=1}^{m}\beta_{i,j}}\sum_{i=1}^{m}\beta_{i,j}\left(x^{(i)}-\mu_j\right)^T\left(x^{(i)}-\mu_j\right)$$

$$
\begin{aligned}
Likelihood\ \ell(\theta|X) &= P(X|\theta) \\
&\propto p(X|\theta) \\
&\propto \prod_{i=1}^{m} p\left(x^{(i)}|\theta\right)
\end{aligned}
$$

$$
\begin{aligned}
Log\ likelihood\ \mathfrak{L}(\theta|X) &= \ln\left(\ell(\theta|X)\right) \\
&\propto \ln\left(\prod_{i=1}^{m} p\left(x^{(i)}|\theta\right)\right) \\
&\propto \sum_{i=1}^{m} \ln\left(p\left(x^{(i)}|\theta\right)\right)
\end{aligned}
$$

The objective is,

$$
\begin{aligned}
\hat{\theta} &= \arg\max_{\theta\in\Theta} \ell(\theta|X)\ \ subject\ to \sum_{j=1}^{c} \alpha_j = 1 \\
&\propto \arg\max_{\theta\in\Theta} \mathfrak{L}(\theta|X)\ \ subject\ to \sum_{j=1}^{c} \alpha_j = 1
\end{aligned}
$$

where Θ is the set of all possible θ.

$$
let\ g(\alpha_1, \dots, \alpha_c) = \sum_{j=1}^{c} \alpha_j - 1 = 1
$$

By Lagrange Multiplier,

$$
\nabla\mathfrak{L}(\theta|X) = \lambda g(\alpha_1, \dots, \alpha_c)
$$

$$
\mathcal{L}(X, \theta) = \mathfrak{L}(\theta|X) - \lambda g(\alpha_1, \dots, \alpha_c)
$$

$$
\begin{aligned}
\frac{\partial \mathcal{L}(X,\theta)}{\partial \alpha_w} &= \frac{\partial}{\partial \alpha_w}\left(\mathfrak{L}(\theta|X) - \lambda g(\alpha_1, \dots, \alpha_c)\right) \\
&= \frac{\partial \sum_{i=1}^{m} \ln\left(p\left(x^{(i)}|\theta\right)\right)}{\partial \alpha_w} - \frac{\partial \lambda \sum_{j=1}^{c} \alpha_j}{\partial \alpha_w} + \frac{\partial \lambda}{\partial \alpha_w} \\
&= \frac{\partial \sum_{i=1}^{m} \ln\left[\sum_{j=1}^{c} \alpha_j \frac{1}{\sqrt{(2\pi)^n|\Sigma|}} e^{\left[-\frac{1}{2}\left(x^{(i)}-\mu_j\right)^T \Sigma_j^{-1}\left(x^{(i)}-\mu_j\right)\right]}\right]}{\partial \alpha_w} - \lambda \\
&= \sum_{i=1}^{m} \frac{\partial \ln\left[\sum_{j=1}^{c} \alpha_j \frac{1}{\sqrt{(2\pi)^n|\Sigma|}} e^{\left[-\frac{1}{2}\left(x^{(i)}-\mu_j\right)^T \Sigma_j^{-1}\left(x^{(i)}-\mu_j\right)\right]}\right]}{\partial \alpha_w} - \lambda \\
&= \sum_{i=1}^{m} \frac{\partial \ln\left[\sum_{j=1}^{c} \alpha_j \mathcal{N}_j\left(x^{(i)}|\mu_j, \Sigma_j\right)\right]}{\partial \sum_{j=1}^{c} \alpha_j \mathcal{N}_j\left(x^{(i)}|\mu_j, \Sigma_j\right)} \frac{\partial \sum_{j=1}^{c} \alpha_j \mathcal{N}_j\left(x^{(i)}|\mu_j, \Sigma_j\right)}{\partial \alpha_w} - \lambda \\
&= \sum_{i=1}^{m} \frac{\mathcal{N}_w\left(x^{(i)}|\mu_w, \Sigma_w\right)}{\sum_{j=1}^{c} \alpha_j \mathcal{N}_j\left(x^{(i)}|\mu_j, \Sigma_j\right)} - \lambda
\end{aligned}
$$

$$
Set\ \frac{\partial \mathcal{L}(X,\theta)}{\partial \alpha_w} = 0
$$

$$\sum_{i=1}^{m} \frac{\mathcal{N}_w\left(x^{(i)}|\mu_w, \Sigma_w\right)}{\sum_{j=1}^{c} \alpha_j \mathcal{N}_j\left(x^{(i)}|\mu_j, \Sigma_j\right)} - \lambda = 0$$

$$\sum_{i=1}^{m} \frac{\mathcal{N}_w\left(x^{(i)}|\mu_w, \Sigma_w\right)}{\sum_{j=1}^{c} \alpha_j \mathcal{N}_j\left(x^{(i)}|\mu_j, \Sigma_j\right)} = \lambda$$

$$\sum_{i=1}^{m} \frac{\alpha_w \mathcal{N}_w\left(x^{(i)}|\mu_w, \Sigma_w\right)}{\sum_{j=1}^{c} \alpha_j \mathcal{N}_j\left(x^{(i)}|\mu_j, \Sigma_j\right)} = \alpha_w \lambda$$

$$\sum_{i=1}^{m} \beta_{i,w} = \alpha_w \lambda$$

$$\alpha_w = \frac{\sum_{i=1}^{m} \beta_{i,w}}{\lambda}$$

$$\because \sum_{j=1}^{c} \alpha_j = 1$$

$$\sum_{j=1}^{c} \frac{\sum_{i=1}^{m} \beta_{i,j}}{\lambda} = 1$$

$$\frac{1}{\lambda} \sum_{j=1}^{c} \sum_{i=1}^{m} \beta_{i,j} = 1$$

$$\sum_{j=1}^{c} \sum_{i=1}^{m} \beta_{i,j} = \lambda$$

$$\sum_{i=1}^{m} \sum_{j=1}^{c} \beta_{i,j} = \lambda$$

$$\sum_{i=1}^{m} 1 = \lambda$$

$$\lambda = m$$

$$\therefore \alpha_w = \frac{1}{m} \sum_{i=1}^{m} \beta_{i,w}$$

Question 3:

To solve the problem analytically, we have to use Lagrange Multiplier.

$$f(x, y) = x + y$$

$$\max f(x, y) \ \ subject\ to\ x^2 + y^2 = 1$$

$$let\ g(x, y) = x^2 + y^2 - 1 = 0$$

$$\because \nabla f(x, y) = \lambda \nabla g(x, y)$$

$$\begin{aligned} \therefore \mathcal{L}(x, y, \lambda) &= f(x, y) - \lambda g(x, y) \\ &= x + y - \lambda(x^2 + y^2 - 1) \\ &= x + y - \lambda x^2 - \lambda y^2 + \lambda \end{aligned}$$

$$\because Find\ maximum\ point$$

$$\therefore Set\ \nabla\mathcal{L} = \vec{\mathbf{0}}$$

$$\frac{\partial\mathcal{L}}{\partial x} = 1 - 2\lambda x = 0$$
$$\frac{\partial\mathcal{L}}{\partial y} = 1 - 2\lambda y = 0$$
$$\frac{\partial\mathcal{L}}{\partial \lambda} = -x^2 - y^2 + 1 = 0$$

$$\because By\ substitution\ within\ the\ simutaneuous\ equations$$

$$x = \frac{1}{\sqrt{2}}\ or\ -\frac{1}{\sqrt{2}}$$
$$y = \frac{1}{\sqrt{2}}\ or\ -\frac{1}{\sqrt{2}}$$
$$\lambda = \frac{1}{\sqrt{2}}\ or\ -\frac{1}{\sqrt{2}}$$

$$when\ x = y = \lambda = \frac{1}{\sqrt{2}}, f(x,y) \approx 1.4142135623730951$$

$$when\ x = y = \lambda = -\frac{1}{\sqrt{2}}, f(x,y) \approx -1.4142135623730951$$

$$\therefore x = y = \lambda = \frac{1}{\sqrt{2}}\ will\ lead\ to\ maximum\ point\ of f(x,y)\ subject\ to\ x^2 + y^2 = 1$$

Then, we use try to use numerical approximation with gradient descent to figure out the point $\max f(x,y)\ subject\ to\ x^2 + y^2 = 1$. The code is uploaded to

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW4_Q3.ipynb

Since the gradient search coded is set to be running for 100000 epochs, the standard output is so tremendous that it can't be shown via GitHub directly.

I've tried three kinds of initialization,

1. $x = y = \lambda = \frac{1}{\sqrt{2}}$
2. $x = y = \lambda = random\ initialization\ with\ normal\ distribution$
3. $x = y = \frac{1}{\sqrt{2}}, \lambda = 0.707 \quad which\ is\ slightly\ smaller\ than\ \frac{1}{\sqrt{2}}$

the results will be discussed later.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
learning_rate = 0.0001
epoch = 100000
x = np.random.randn()
y = np.random.randn()
lamb = np.random.randn()
```

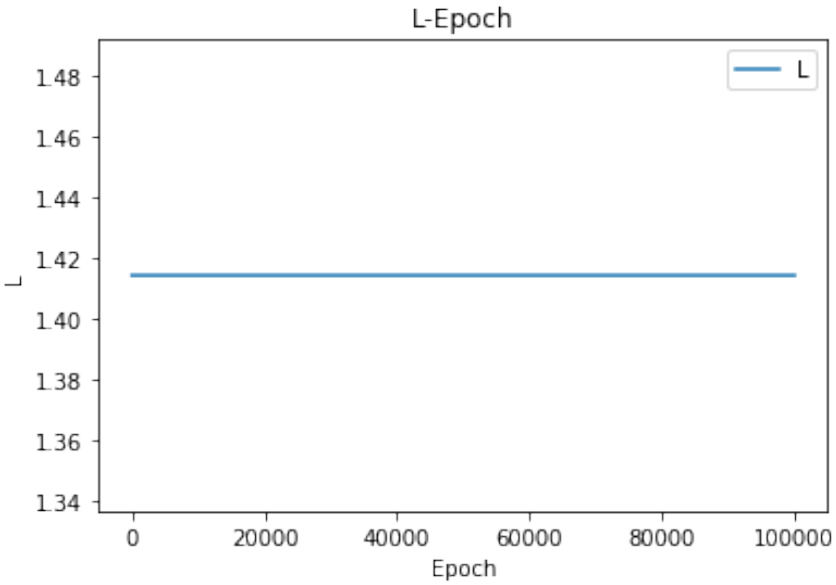
```
L_history = list()
f = lambda x, y: x + y
g = lambda x, y: x**2 + y**2 - 1
l = lambda x, y, lamb: f(x, y) - lamb * g(x, y)
L_history.append(l(x, y, lamb))
print('Initial State: L: {}, x: {}, y: {}, lambda: {}'.format(L_history[0], x, y, lamb))
for i in range(epoch):
    temp_x = x + learning_rate * (1 - 2 * lamb * x)
    temp_y = y + learning_rate * (1 - 2 * lamb * y)
    temp_lamb = lamb + learning_rate * (-x**2 - y**2 + 1)
    x = temp_x
    y = temp_y
    lamb = temp_lamb
    L_history.append(l(x, y, lamb))
    print('Epoch {}: L: {}, x: {}, y: {}, lambda: {}'.format(i + 1, L_history[i + 1], x, y,
lamb))
```

```
plt.plot(L_history, label='L')
plt.title('L-Epoch')
plt.legend(loc='best')
plt.ylabel('L')
plt.xlabel('Epoch')
plt.show()
```

For the initialization of $x = y = \lambda = \frac{1}{\sqrt{2}}$,

```
Initial State: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 1: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 2: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 3: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 4: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 5: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 6: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 7: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 8: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 9: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 10: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 11: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 12: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 13: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 14: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 15: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 16: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 17: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 18: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475

Epoch 99986: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99987: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99988: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99989: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99990: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99991: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99992: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99993: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99994: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99995: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99996: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99997: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99998: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 99999: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
Epoch 100000: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.7071067811865475
```



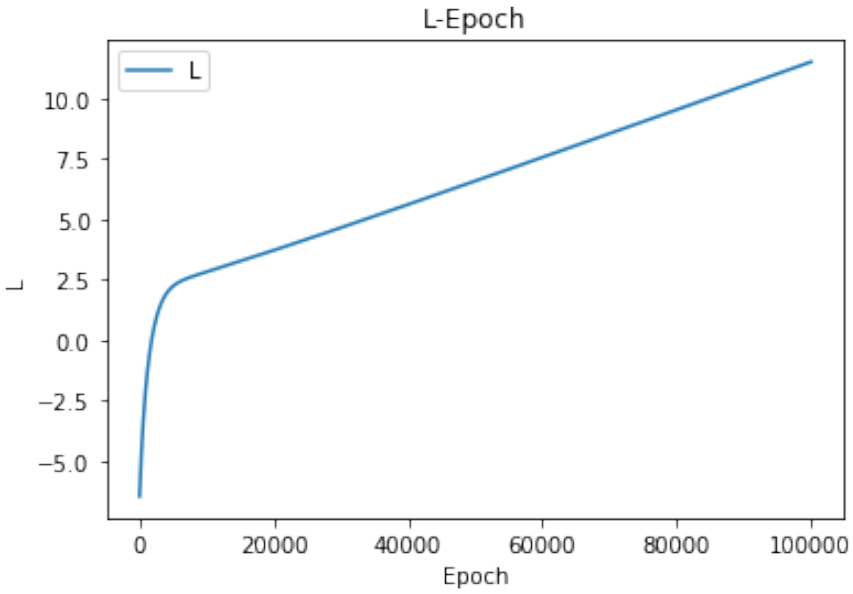
We can see the convergence of $\mathcal{L}$, x, y, λ remain the same value since $\frac{1}{\sqrt{2}}$ is the point that derived analytically.

For the initialization of

$x = y = \lambda = random\ initialization\ with\ normal\ distribution$,

```
Initial State: L: -6.462397297271805, x: -0.5262718561196011, y: -1.691762888605743, lambda: 1.984252197639234
Epoch 1: L: -6.455037639504784, x: -0.525963004902189, y: -1.6909915117598229, lambda: 1.9840382952654532
Epoch 2: L: -6.4476859664621475, x: -0.5256542987534651, y: -1.6902205133765629, lambda: 1.9838246863279163
Epoch 3: L: -6.440342266372607, x: -0.5253457375585968, y: -1.6894498931406081, lambda: 1.9836113705453526
Epoch 4: L: -6.433006527486461, x: -0.525037321202899, y: -1.688679650737008, lambda: 1.9833983476368122
Epoch 5: L: -6.4256787380755735, x: -0.5247290495718347, y: -1.6879097858512162, lambda: 1.9831856173216653
Epoch 6: L: -6.418358886433304, x: -0.5244209225510144, y: -1.6871402981690888, lambda: 1.9829731793196017
Epoch 7: L: -6.411046960874481, x: -0.5241129400261958, y: -1.6863711873768852, lambda: 1.9827610333506303
Epoch 8: L: -6.403742949735342, x: -0.523805101883284, y: -1.685602453161266, lambda: 1.9825491791350784
Epoch 9: L: -6.396446841373495, x: -0.5234974080083309, y: -1.6848340952092933, lambda: 1.9823376163935922
Epoch 10: L: -6.389158624167875, x: -0.523189858287535, y: -1.68406611320843, lambda: 1.9821263448471351
Epoch 11: L: -6.38187828651869, x: -0.5228824526072412, y: -1.6832985068465391, lambda: 1.981915364216988
Epoch 12: L: -6.374605816847385, x: -0.5225751908539409, y: -1.6825312758118827, lambda: 1.9817046742247484
Epoch 13: L: -6.36734120359659, x: -0.522268072914271, y: -1.6817644197931214, lambda: 1.9814942745923303
Epoch 14: L: -6.36008443523008, x: -0.5219610986750146, y: -1.6809979384793148, lambda: 1.9812841650419635
Epoch 15: L: -6.352835500232729, x: -0.5216542680231001, y: -1.6802318315599194, lambda: 1.9810743452961932
Epoch 16: L: -6.345594387110461, x: -0.5213475808456012, y: -1.6794660987247887, lambda: 1.9808648150778798
Epoch 17: L: -6.338361084390209, x: -0.5210410370297366, y: -1.6787007396641727, lambda: 1.9806555741101979
Epoch 18: L: -6.331135580619872, x: -0.52073463646287, y: -1.677935754068717, lambda: 1.9804466221166361

Epoch 99982: L: 11.525536683530875, x: 0.043712872115687036, y: 0.043712872115687036, lambda: 11.481990864377595
Epoch 99983: L: 11.52563592358951, x: 0.04371248995602943, y: 0.04371248995602943, lambda: 11.482090482214558
Epoch 99984: L: 11.525735163661354, x: 0.043712107803053825, y: 0.043712107803053825, lambda: 11.482190100058203
Epoch 99985: L: 11.525834403746407, x: 0.04371172565676004, y: 0.04371172565676004, lambda: 11.482289717908529
Epoch 99986: L: 11.525933643844674, x: 0.043711343517147914, y: 0.043711343517147914, lambda: 11.482389335765538
Epoch 99987: L: 11.52603288395615, x: 0.043710961384217255, y: 0.043710961384217255, lambda: 11.482488953629227
Epoch 99988: L: 11.526132124080831, x: 0.043710579257967896, y: 0.043710579257967896, lambda: 11.482588571499598
Epoch 99989: L: 11.526231364218726, x: 0.043710197138399666, y: 0.043710197138399666, lambda: 11.482688189376649
Epoch 99990: L: 11.526330604369829, x: 0.04370981502551238, y: 0.04370981502551238, lambda: 11.482787807260383
Epoch 99991: L: 11.526429844534139, x: 0.043709432919305874, y: 0.043709432919305874, lambda: 11.482887425150798
Epoch 99992: L: 11.526529084711658, x: 0.04370905081977996, y: 0.04370905081977996, lambda: 11.482987043047892
Epoch 99993: L: 11.526628324902383, x: 0.04370866872693447, y: 0.04370866872693447, lambda: 11.483086660951667
Epoch 99994: L: 11.526727565106318, x: 0.04370828664076923, y: 0.04370828664076923, lambda: 11.483186278862123
Epoch 99995: L: 11.52682680532346, x: 0.04370790456128406, y: 0.04370790456128406, lambda: 11.48328589677926
Epoch 99996: L: 11.526926045553807, x: 0.043707522488478784, y: 0.043707522488478784, lambda: 11.483385514703075
Epoch 99997: L: 11.527025285797361, x: 0.043707140422353234, y: 0.043707140422353234, lambda: 11.48348513263357
Epoch 99998: L: 11.527124526054122, x: 0.04370675836290723, y: 0.04370675836290723, lambda: 11.483584750570746
Epoch 99999: L: 11.527223766324088, x: 0.0437063763101406, y: 0.0437063763101406, lambda: 11.4836843685146
Epoch 100000: L: 11.52732300660726, x: 0.043705994264053166, y: 0.043705994264053166, lambda: 11.483783986465134
```

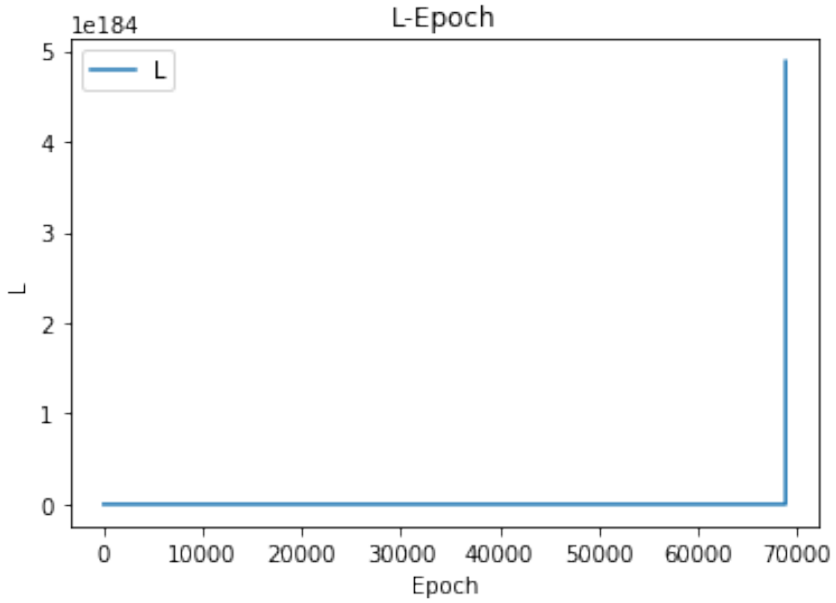


We can see the divergence.

For the initialization of $x = y = \frac{1}{\sqrt{2}}, \lambda = 0.707$,

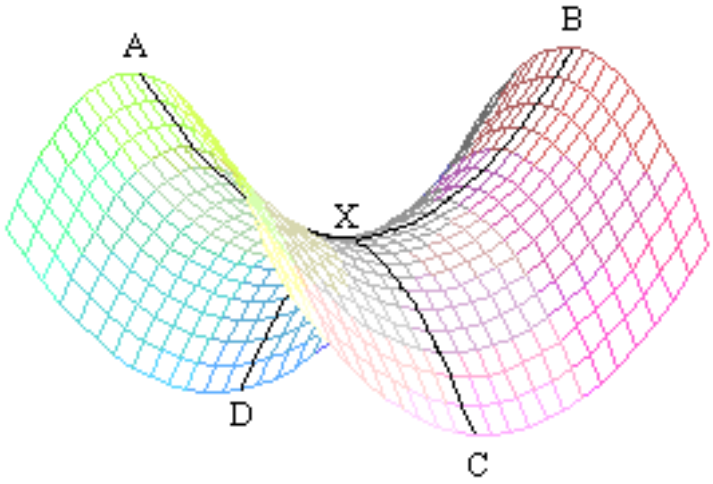
```
Initial State: L: 1.4142135623730951, x: 0.7071067811865475, y: 0.7071067811865475, lambda: 0.707
Epoch 1: L: 1.4142135623776557, x: 0.7071067962876877, y: 0.7071067962876877, lambda: 0.707
Epoch 2: L: 1.4142135623822147, x: 0.7071068113866926, y: 0.7071068113866926, lambda: 0.7069999999957287
Epoch 3: L: 1.4142135623867729, x: 0.7071068264835632, y: 0.7071068264835632, lambda: 0.7069999999871868
Epoch 4: L: 1.4142135623913297, x: 0.7071068415783002, y: 0.7071068415783002, lambda: 0.7069999999743748
Epoch 5: L: 1.414213562395885, x: 0.7071068566709047, y: 0.7071068566709047, lambda: 0.7069999999572935
Epoch 6: L: 1.414213562400439, x: 0.7071068717613774, y: 0.7071068717613774, lambda: 0.7069999999359432
Epoch 7: L: 1.414213562404992, x: 0.7071068868497195, y: 0.7071068868497195, lambda: 0.7069999999103248
Epoch 8: L: 1.4142135624095435, x: 0.7071069019359316, y: 0.7071069019359316, lambda: 0.7069999998804387
Epoch 9: L: 1.414213562414094, x: 0.7071069170200148, y: 0.7071069170200148, lambda: 0.7069999998462856
Epoch 10: L: 1.414213562418643, x: 0.70710693210197, y: 0.70710693210197, lambda: 0.7069999998078661
Epoch 11: L: 1.4142135624231906, x: 0.7071069471817979, y: 0.7071069471817979, lambda: 0.7069999997651808
Epoch 12: L: 1.414213562427737, x: 0.7071069622594996, y: 0.7071069622594996, lambda: 0.7069999997182302
Epoch 13: L: 1.4142135624322822, x: 0.707106977335076, y: 0.707106977335076, lambda: 0.7069999996670151
Epoch 14: L: 1.414213562436826, x: 0.7071069924085279, y: 0.7071069924085279, lambda: 0.7069999996115359
Epoch 15: L: 1.4142135624413688, x: 0.7071070074798563, y: 0.7071070074798563, lambda: 0.7069999995517933
Epoch 16: L: 1.41421356244591, x: 0.7071070225490621, y: 0.7071070225490621, lambda: 0.7069999994877878
Epoch 17: L: 1.4142135624504502, x: 0.7071070376161461, y: 0.7071070376161461, lambda: 0.7069999994195202
Epoch 18: L: 1.414213562454989, x: 0.7071070526811093, y: 0.7071070526811093, lambda: 0.706999999346991
```

```
Epoch 99982: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99983: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99984: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99985: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99986: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99987: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99988: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99989: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99990: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99991: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99992: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99993: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99994: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99995: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99996: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99997: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99998: L: inf, x: inf, y: inf, lambda: -inf
Epoch 99999: L: inf, x: inf, y: inf, lambda: -inf
Epoch 100000: L: inf, x: inf, y: inf, lambda: -inf
```



We can see that the serious divergence is occurred.

In short, we can conjecture that there are saddle points in $\mathcal{L}$.



Assume the above saddle point is occurred in $\mathcal{L}$, the point X should be $x = y = \lambda = \frac{1}{\sqrt{2}}$ such that $\nabla\mathcal{L} = \vec{\mathbf{0}}$. Nevertheless, the gradient search algorithm will easily pass the point X and will directly go up towards A or B in this case. Thus, using gradient search in solving constrained optimization problem transformed by Lagrange Multiplier will not easily give us the desired result since the solving of constrained optimization problem transformed by Lagrange Multiplier should be an analytical process.

Question 4:

Assume we have only one single tuple of data, that is, we omit the summation along the data for simplicity. The lost function is Mean Squared Error,

$$L_i(d_i, y_i | x_1, x_2, w_1, \dots, w_8) = \frac{1}{2}(y_i - d_i)^2$$

$$C(d_1, y_1, d_2, y_2 | x_1, x_2, w_1, \dots, w_8) = L_1 + L_2$$

We have to firstly find the gradient of the cost function,

$$\nabla C_{w_{1\sim8}} = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ \frac{\partial C}{\partial w_3} \\ \frac{\partial C}{\partial w_4} \\ \frac{\partial C}{\partial w_5} \\ \frac{\partial C}{\partial w_6} \\ \frac{\partial C}{\partial w_7} \\ \frac{\partial C}{\partial w_8} \end{bmatrix}$$

We can do that by using chain rule. Let we consider all the weights at time t.

$$\because sigmoid\ used$$

$$\frac{\partial C}{\partial w_1} = \frac{\partial \frac{1}{2}(y_1 - d_1)^2}{\partial (y_1 - d_1)} \frac{\partial (y_1 - d_1)}{\partial y_1} \frac{\partial y_1}{\partial z_1} \frac{\partial z_1}{\partial h_1} \frac{\partial h_1}{\partial q_1} \frac{\partial q_1}{\partial w_1} + \frac{\partial \frac{1}{2}(y_2 - d_2)^2}{\partial (y_2 - d_2)} \frac{\partial (y_2 - d_2)}{\partial y_2} \frac{\partial y_2}{\partial z_2} \frac{\partial z_2}{\partial h_1} \frac{\partial h_1}{\partial q_1} \frac{\partial q_1}{\partial w_1}$$

$$\frac{\partial C}{\partial w_1} = (y_1 - d_1)(1)\big(y_1(1 - y_1)\big)(w_5)\big(h_1(1 - h_1)\big)(x_1) + (y_2 - d_2)(1)\big(y_2(1 - y_2)\big)(w_7)\big(h_1(1 - h_1)\big)(x_1)$$

$$\frac{\partial C}{\partial w_2} = (y_1 - d_1)(1)\big(y_1(1 - y_1)\big)(w_5)\big(h_1(1 - h_1)\big)(x_2) + (y_2 - d_2)(1)\big(y_2(1 - y_2)\big)(w_7)\big(h_1(1 - h_1)\big)(x_2)$$

$$\frac{\partial C}{\partial w_3} = (y_1 - d_1)(1)\big(y_1(1 - y_1)\big)(w_6)\big(h_2(1 - h_2)\big)(x_1) + (y_2 - d_2)(1)\big(y_2(1 - y_2)\big)(w_8)\big(h_2(1 - h_2)\big)(x_1)$$

$$\frac{\partial C}{\partial w_4} = (y_1 - d_1)(1)\big(y_1(1 - y_1)\big)(w_6)\big(h_2(1 - h_2)\big)(x_2) + (y_2 - d_2)(1)\big(y_2(1 - y_2)\big)(w_8)\big(h_2(1 - h_2)\big)(x_2)$$

$$\frac{\partial C}{\partial w_5} = (y_1 - d_1)(1)\big(y_1(1 - y_1)\big)(h_1)$$

$$\frac{\partial C}{\partial w_6} = (y_1 - d_1)(1)\big(y_1(1 - y_1)\big)(h_2)$$

$$\frac{\partial C}{\partial w_7} = (y_2 - d_2)(1)\big(y_2(1-y_2)\big)(h_1)$$

$$\frac{\partial C}{\partial w_8} = (y_2 - d_2)(1)\big(y_2(1-y_2)\big)(h_2)$$

After computing all the derivative terms, the update rule for weights is

$$W_{t+1} = W_t - \alpha \nabla C_{w_{1\sim 8}}$$

$$\begin{bmatrix} w_{1,t+1} \\ w_{2,t+1} \\ w_{3,t+1} \\ w_{4,t+1} \\ w_{5,t+1} \\ w_{6,t+1} \\ w_{7,t+1} \\ w_{8,t+1} \end{bmatrix} = \begin{bmatrix} w_{1,t} \\ w_{2,t} \\ w_{3,t} \\ w_{4,t} \\ w_{5,t} \\ w_{6,t} \\ w_{7,t} \\ w_{8,t} \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial C}{\partial w_{1,t}} \\ \frac{\partial C}{\partial w_{2,t}} \\ \frac{\partial C}{\partial w_{3,t}} \\ \frac{\partial C}{\partial w_{4,t}} \\ \frac{\partial C}{\partial w_{5,t}} \\ \frac{\partial C}{\partial w_{6,t}} \\ \frac{\partial C}{\partial w_{7,t}} \\ \frac{\partial C}{\partial w_{8,t}} \end{bmatrix}$$

where α is the learning rate.

Question 5:

The whole neural networks and back propagation are implemented and uploaded to

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW4_Q5.ipynb

Since the amount of data of the question required is so small that the average accuracy with cross validation of 10 times is not enough to give us a stable insight. I set the cross validation times to 100.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
"""
Load UCI ML Iris data
Return: data(shape = (150, 4)) and labels(shape = (150, 1)) in numpy array(rank 2)
"""
def load_data(classes):
    data = pd.read_csv('Iris.csv', index_col=0).as_matrix()
    features = data[:, :-1]
    labels = data[:, -1].reshape(-1, 1)
    for class_id in classes:
```

```
        labels[labels == class_id[0]] = class_id[1]
    return features.astype(np.float32), labels.astype(np.float32)
"""
Perform Data Shuffling
Return: shuffled features and labels
"""
def data_shuffling(features, labels):
    shuffle_array = np.random.permutation(features.shape[0])
    features = features[shuffle_array]
    labels = labels[shuffle_array]
    return features, labels
"""
Splitting the data into train and test set
Variable: ratio = percentage of data for train set
Return: train set and test set
"""
def data_split(features, labels, ratio=0.7):
    train_end_index = np.ceil(features.shape[0] * 0.7).astype(np.int32)
    train_features = features[:train_end_index]
    train_labels = labels[:train_end_index]
    test_features = features[train_end_index:]
    test_labels = labels[train_end_index:]
    return train_features, train_labels, test_features, test_labels
"""
Perform standardization for features
Return: standardized features, standard deviation, mean
"""
def standardization(features, exist_params=False, std=None, mean=None, colvar=True):
    features = features.T if not colvar else features
    if not exist_params:
        std = np.std(features, axis=0, keepdims=True)
        mean = np.mean(features, axis=0, keepdims=True)
    standard_features = (features - mean) / std
    return standard_features, std, mean
"""
Preparation of data, Performing shuffling, splitting, one-hotting
"""
def data_preparation(data, labels, classes):
    labels = np.eye(len(classes), dtype=np.float32)[np.squeeze(labels).astype(int)]
    data, labels = data_shuffling(data, labels)
    train_data = np.empty((0, data.shape[1]), dtype=np.float32)
    train_labels = np.empty((0, labels.shape[1]), dtype=np.float32)
    validation_data = np.empty((0, data.shape[1]), dtype=np.float32)
    validation_labels = np.empty((0, labels.shape[1]), dtype=np.float32)
    for each_class in classes:
        data_class = data[np.argmax(labels, axis=1).astype(np.float32) ==
float(each_class[1])]
        labels_class = labels[np.argmax(labels, axis=1).astype(np.float32) ==
float(each_class[1])]
        a, b, c, d = data_split(data_class, labels_class)
        train_data = np.append(train_data, a, axis=0)
        train_labels = np.append(train_labels, b, axis=0)
        validation_data = np.append(validation_data, c, axis=0)
        validation_labels = np.append(validation_labels, d, axis=0)
    train_data, train_labels = data_shuffling(train_data, train_labels)
    validation_data, validation_labels = data_shuffling(validation_data, validation_labels)
    train_data, std, mean = standardization(train_data)
    validation_data, _, _ = standardization(validation_data, exist_params=True, std=std,
mean=mean)
    return train_data, train_labels, validation_data, validation_labels
"""
Sigmoid function
"""
```

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z).astype(np.float32))
"""
Define the architecture of neural networks
"""
def neural_networks(x, w1, w2, b1, b2):
    Z1 = np.dot(x, w1) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(A1, w2) + b2
    A2 = sigmoid(Z2)
    return Z1, A1, Z2, A2
"""
Define the cost function MSE
"""
def mse(outputs, labels_smooth):
    return np.sum(np.sum(np.square((outputs - labels_smooth)) / 2, axis=0) /
outputs.shape[0]) / outputs.shape[1]
"""
Compute the accuracy
"""
def get_accuracy(predictions, ground_truth):
    return np.mean(np.squeeze(predictions == ground_truth).astype(np.float32))
"""
Training of neural networks
"""
def training(train_data, train_labels, validation_data, validation_labels, learning_rate=0.1,
epochs=3500):
    #Label smoothing#
    train_labels_smooth = np.copy(train_labels)
    validation_labels_smooth = np.copy(validation_labels)
    train_labels_smooth[train_labels_smooth == 1.] = 0.9
    train_labels_smooth[train_labels_smooth == 0.] = 0.1
    validation_labels_smooth[validation_labels_smooth == 1.] = 0.9
    validation_labels_smooth[validation_labels_smooth == 0.] = 0.1

    #Initialize the weights and bias#
    w1 = np.random.randn(2, 2).astype(np.float32)
    w2 = np.random.randn(2, 2).astype(np.float32)
    b1 = np.zeros((1, 2))
    b2 = np.zeros((1, 2))

    #Accuracy and cost history#
    accs = list()
    costs = list()
    accs_val = list()
    costs_val = list()

    #Save the initial training cost and accuracy#
    Z1, A1, Z2, A2 = neural_networks(train_data, w1, w2, b1, b2)
    costs.append(mse(A2, train_labels_smooth))
    accs.append(get_accuracy(np.argmax(A2, axis=1), np.argmax(train_labels, axis=1)))

    #Save the initial validation cost and accuracy#
    Z1, A1, Z2, A2 = neural_networks(validation_data, w1, w2, b1, b2)
    costs_val.append(mse(A2, validation_labels_smooth))
    accs_val.append(get_accuracy(np.argmax(A2, axis=1), np.argmax(validation_labels,
axis=1)))

    #Training#
    for i in range(epochs):
        #Feed Forward#
        Z1, A1, Z2, A2 = neural_networks(train_data, w1, w2, b1, b2)
```

```
        #Back Propagation and Update weights, bias#
        dZ2 = np.multiply(np.multiply((A2 - train_labels_smooth), A2), (1 - A2))
        dZ1 = np.multiply(np.dot(dZ2, w2.T), np.multiply(A1, (1 - A1)))
        dw2 = np.dot(A1.T, dZ2) / train_data.shape[0]
        dw1 = np.dot(train_data.T, dZ1) / train_data.shape[0]
        db2 = np.sum(dZ2, axis=0)  / train_data.shape[0]
        db1 = np.sum(dZ1, axis=0) / train_data.shape[0]
        w1 = w1 - learning_rate * dw1
        w2 = w2 - learning_rate * dw2
        b1 = b1 - learning_rate * db1
        b2 = b2 - learning_rate * db2

        #Save the initial training cost and accuracy#
        Z1, A1, Z2, A2 = neural_networks(train_data, w1, w2, b1, b2)
        costs.append(mse(A2, train_labels_smooth))
        accs.append(get_accuracy(np.argmax(A2, axis=1), np.argmax(train_labels, axis=1)))

        #Save the initial validation cost and accuracy#
        Z1, A1, Z2, A2 = neural_networks(validation_data, w1, w2, b1, b2)
        costs_val.append(mse(A2, validation_labels_smooth))
        accs_val.append(get_accuracy(np.argmax(A2, axis=1), np.argmax(validation_labels,
axis=1)))
    return w1, w2, b1, b2, costs, accs, costs_val, accs_val
```

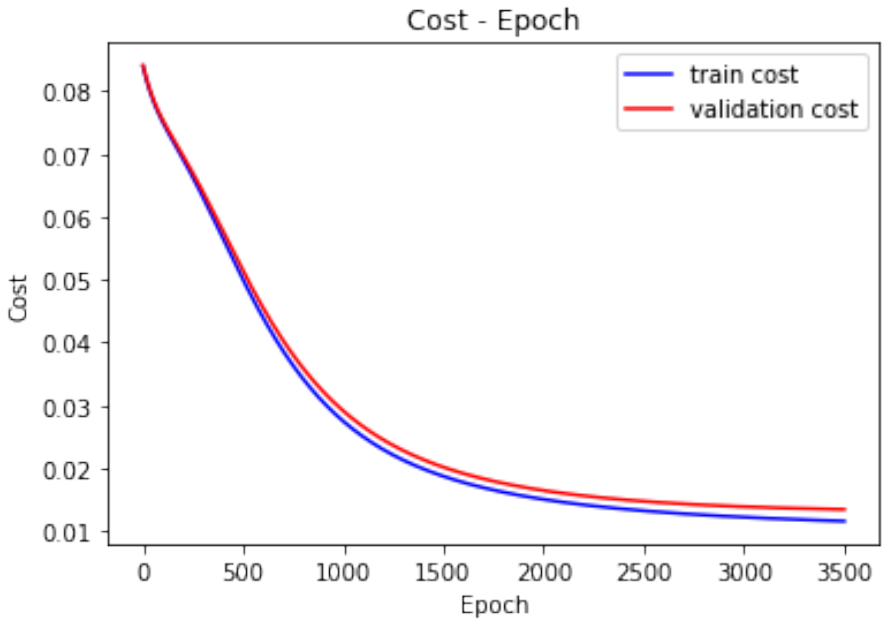
```
"""
Plot the scatter graphs
"""
def plot_scatter(features, labels):
    plt.plot(features[np.squeeze(labels == float(1))][:, 0], features[np.squeeze(labels ==
float(1))][:, 1], 'gx', label='Iris-versicolor')
    plt.plot(features[np.squeeze(labels == float(2))][:, 0], features[np.squeeze(labels ==
float(2))][:, 1], 'bx', label='Iris-virginica')
    plt.title('Petal Length - Petal Width')
    plt.legend(loc='best')
    plt.show()
```

```
"""
Settings
"""
classes = [('Iris-setosa', 0), ('Iris-versicolor', 1), ('Iris-virginica', 2)]
cross_val_times = 100
```

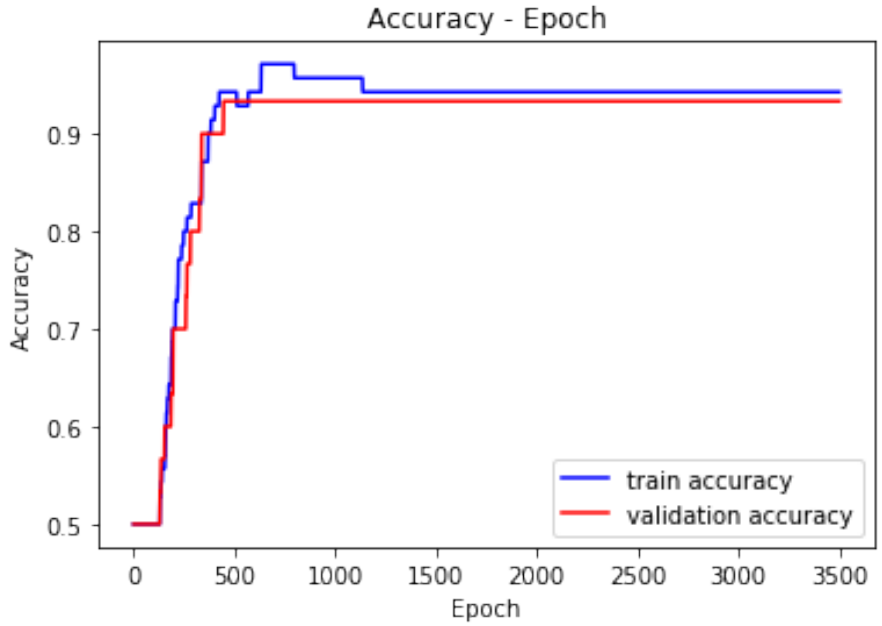
```
data, labels = load_data(classes)
data = data[np.logical_or(np.squeeze(labels == 1), np.squeeze(labels == 2))][:, 2:]
labels = labels[np.logical_or(np.squeeze(labels == 1), np.squeeze(labels == 2))] - 1.
classes = [('Iris-versicolor', 0), ('Iris-virginica', 1)]
cross_val_acc = 0.
for i in range(cross_val_times):
    train_data, train_labels, validation_data, validation_labels = data_preparation(data,
labels, classes)
    w1, w2, b1, b2, costs, accs, costs_val, accs_val = training(train_data, train_labels,
validation_data, validation_labels)
    cross_val_acc += accs_val[-1] / cross_val_times
plt.plot(costs, 'b-', label='train cost')
plt.plot(costs_val, 'r-', label='validation cost')
plt.title('Cost - Epoch')
plt.ylabel('Cost')
plt.xlabel('Epoch')
plt.legend(loc='best')
plt.show()
plt.plot(accs, 'b-', label='train accuracy')
plt.plot(accs_val, 'r-', label='validation accuracy')
plt.title('Accuracy - Epoch')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='best')
```

```
plt.show()
print('Average Accuracy ({} times): {}'.format(cross_val_times, cross_val_acc))
```

The Cost-Epoch graph,



The Accuracy-Epoch graph,



The average accuracy of 100 times is 0.9306666558980935.