# Backpropagation Exercise

In this exercise we will use backpropagation to train a multi-layer perceptron (with a single hidden layer). We will experiment with different patterns and see how quickly or slowly the weights converge. We will see the impact and interplay of different parameters such as learning rate, number of iterations, and number of data points.

# Alunos: Aritana Noara Costa Santos e Victor Augusto Januário da Cruz

In [2]:

```
#Preliminaries
from __future__ import division, print_function
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Fill out the code below so that it creates a multi-layer perceptron with a single hidden layer (with 4 nodes) and trains it via back-propagation. Specifically your code should:

1. Initialize the weights to random values between -1 and 1
2. Perform the feed-forward computation
3. Compute the loss function
4. Calculate the gradients for all the weights via back-propagation
5. Update the weight matrices (using a learning_rate parameter)
6. Execute steps 2-5 for a fixed number of iterations
7. Plot the accuracies and log loss and observe how they change over time

Once your code is running, try it for the different patterns below.

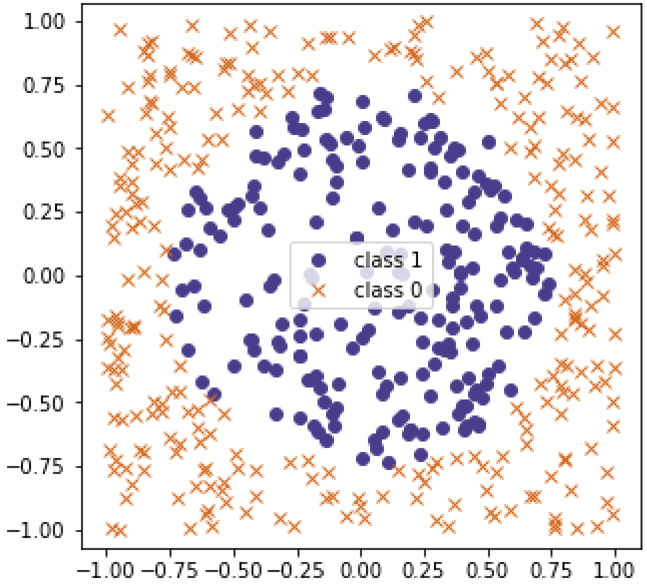
- Which patterns was the neural network able to learn quickly and which took longer?
- What learning rates and numbers of iterations worked well?
- If you have time, try varying the size of the hidden layer and experiment with different activation functions (e.g. ReLu)

## Circle pattern

In [35]:
```python
## This code below generates two x values and a y value according to different patterns
## It also creates a "bias" term (a vector of 1s)
## The goal is then to learn the mapping from x to y using a neural network via back-propagation

import time

num_obs = 500
x_mat_1 = np.random.uniform(-1,1,size = (num_obs,2))
x_mat_bias = np.ones((num_obs,1))
x_mat_full = np.concatenate( (x_mat_1,x_mat_bias), axis=1)

# PICK ONE PATTERN BELOW and comment out the rest.


timer = []

#1 # Circle pattern
start = time.time()
y = (np.sqrt(x_mat_full[:,0]**2 + x_mat_full[:,1]**2)<.75).astype(int)
end = time.time()

timeCircle = round(end - start,7)
timer.append(timeCircle)
print(timer)

print("Circle pattern took",timeCircle,"ms")

print('shape of x_mat_full is {}'.format(x_mat_full.shape))
print('shape of y is {}'.format(y.shape))

fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
# ax.grid(True)
ax.legend(loc='best')
ax.axis('equal');
```

```
[0.0001404]
Circle pattern took 0.0001404 ms
shape of x_mat_full is (500, 3)
shape of y is (500,)
/tmp/ipykernel_11446/230992738.py:32: UserWarning: color is redundantly defined by the 'color' keyword argument and the
fmt string "ro" (-> color='r'). The keyword argument will take precedence.
  ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
```

```
/tmp/ipykernel_11446/230992738.py:33: UserWarning: color is redundantly defined by the 'color' keyword argument and the
fmt string "bx" (-> color='b'). The keyword argument will take precedence.
  ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
```




## Diamond Pattern

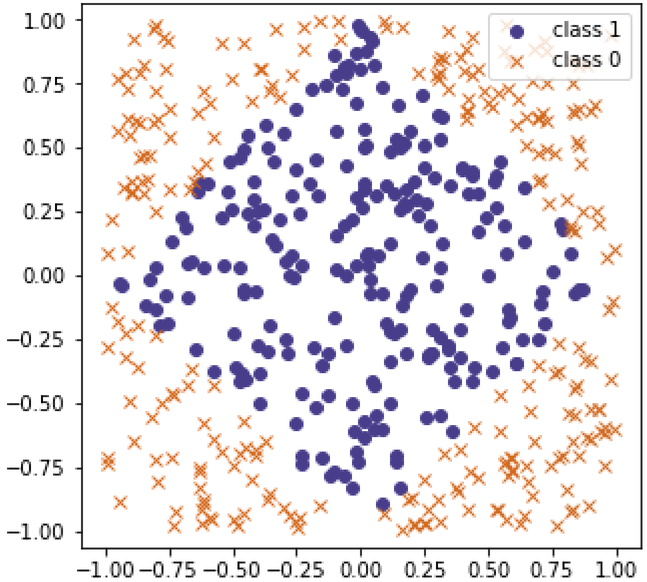
In [36]:

```python
## This code below generates two x values and a y value according to different patterns
## It also creates a "bias" term (a vector of 1s)
## The goal is then to learn the mapping from x to y using a neural network via back-propagation

import time

num_obs = 500
x_mat_1 = np.random.uniform(-1,1,size = (num_obs,2))
x_mat_bias = np.ones((num_obs,1))
x_mat_full = np.concatenate( (x_mat_1,x_mat_bias), axis=1)

#2 # Diamond Pattern
start = time.time()
y = ((np.abs(x_mat_full[:,0]) + np.abs(x_mat_full[:,1]))<1).astype(int)
end = time.time()

timeDiamond = round(end - start,7)
timer.append(timeDiamond)
```

```
print("Diamond Pattern",timeDiamond,"ms")


print('shape of x_mat_full is {}'.format(x_mat_full.shape))
print('shape of y is {}'.format(y.shape))

fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
# ax.grid(True)
ax.legend(loc='best')
ax.axis('equal');
```

```
Diamond Pattern 0.0002389 ms
shape of x_mat_full is (500, 3)
shape of y is (500,)
```

```
/tmp/ipykernel_11446/1277343602.py:26: UserWarning: color is redundantly defined by the 'color' keyword argument and th
e fmt string "ro" (-> color='r'). The keyword argument will take precedence.
  ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
/tmp/ipykernel_11446/1277343602.py:27: UserWarning: color is redundantly defined by the 'color' keyword argument and th
e fmt string "bx" (-> color='b'). The keyword argument will take precedence.
  ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
```

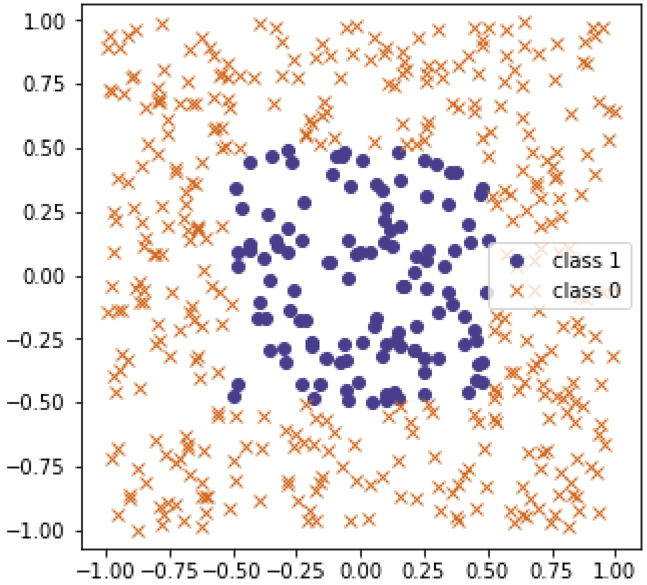



## Escolha para questão 02: Centered square

In [60]:
```
## This code below generates two x values and a y value according to different patterns
## It also creates a "bias" term (a vector of 1s)
## The goal is then to learn the mapping from x to y using a neural network via back-propagation

import time

num_obs = 500
x_mat_1 = np.random.uniform(-1,1,size = (num_obs,2))
x_mat_bias = np.ones((num_obs,1))
x_mat_full = np.concatenate( (x_mat_1,x_mat_bias), axis=1)


#3 # Centered square
start = time.time()
y = ((np.maximum(np.abs(x_mat_full[:,0]), np.abs(x_mat_full[:,1])))<.5).astype(int)
end = time.time()

timeCenteredSquare = round(end - start,7)
timer.append(timeCenteredSquare)

print("Centered square",timeCenteredSquare,"ms")

print('shape of x_mat_full is {}'.format(x_mat_full.shape))
print('shape of y is {}'.format(y.shape))

fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
# ax.grid(True)
ax.legend(loc='best')
ax.axis('equal');
```

```
Centered square 0.0002429 ms
shape of x_mat_full is (500, 3)
shape of y is (500,)
```

```
/tmp/ipykernel_11446/3138982794.py:27: UserWarning: color is redundantly defined by the 'color' keyword argument and th
e fmt string "ro" (-> color='r'). The keyword argument will take precedence.
  ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
/tmp/ipykernel_11446/3138982794.py:28: UserWarning: color is redundantly defined by the 'color' keyword argument and th
e fmt string "bx" (-> color='b'). The keyword argument will take precedence.
  ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
```




## Thick Right Angle pattern

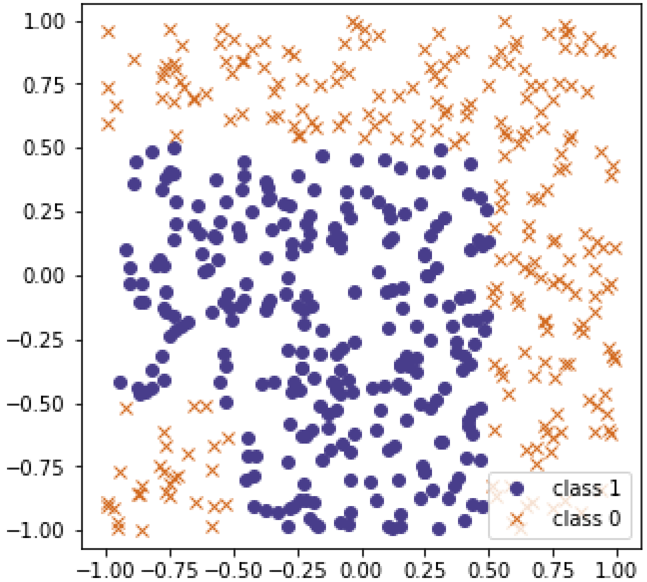
In [31]:

```
## This code below generates two x values and a y value according to different patterns
## It also creates a "bias" term (a vector of 1s)
## The goal is then to learn the mapping from x to y using a neural network via back-propagation

import time

num_obs = 500
x_mat_1 = np.random.uniform(-1,1,size = (num_obs,2))
x_mat_bias = np.ones((num_obs,1))
x_mat_full = np.concatenate( (x_mat_1,x_mat_bias), axis=1)


#4 # Thick Right Angle pattern
start = time.time()
y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>-.5)
end = time.time()

timeThickRightAnglepattern = round(end - start,7)
timer.append(timeThickRightAnglepattern)

print("Thick Right Angle pattern",timeThickRightAnglepattern,"ms")
```

```
print('shape of x_mat_full is {}'.format(x_mat_full.shape))
print('shape of y is {}'.format(y.shape))

fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
# ax.grid(True)
ax.legend(loc='best')
ax.axis('equal');
```

```
Thick Right Angle pattern 0.0002246 ms
shape of x_mat_full is (500, 3)
shape of y is (500,)
```

```
/tmp/ipykernel_11446/1825069254.py:26: UserWarning: color is redundantly defined by the 'color' keyword argument and th
e fmt string "ro" (-> color='r'). The keyword argument will take precedence.
  ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
/tmp/ipykernel_11446/1825069254.py:27: UserWarning: color is redundantly defined by the 'color' keyword argument and th
e fmt string "bx" (-> color='b'). The keyword argument will take precedence.
  ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
```

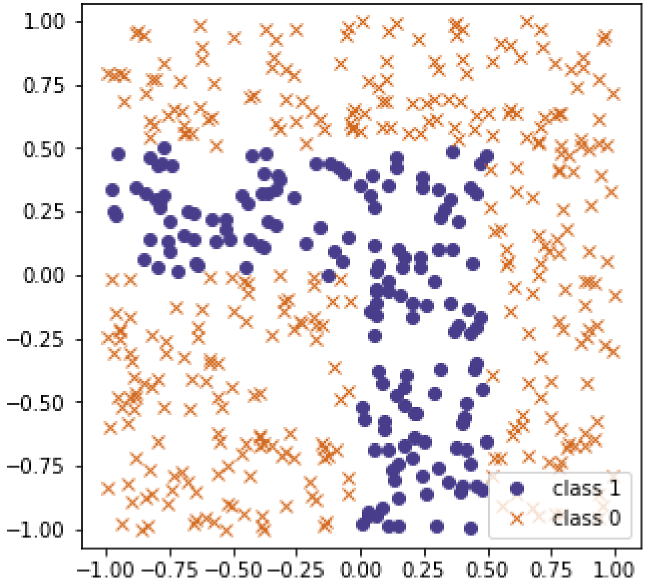



## Thin right angle pattern

In [38]:
```
## This code below generates two x values and a y value according to different patterns
## It also creates a "bias" term (a vector of 1s)
## The goal is then to learn the mapping from x to y using a neural network via back-propagation

import time

num_obs = 500
x_mat_1 = np.random.uniform(-1,1,size = (num_obs,2))
x_mat_bias = np.ones((num_obs,1))
x_mat_full = np.concatenate( (x_mat_1,x_mat_bias), axis=1)


#5 # Thin right angle pattern
start = time.time()
y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>0)).
end = time.time()

timeThinRightAnglePattern = end - start

timeThinRightAnglePattern = round(end - start,7)
timer.append(timeThinRightAnglePattern)

print("Thin right angle pattern",timeThinRightAnglePattern,"ms")

print('shape of x_mat_full is {}'.format(x_mat_full.shape))
print('shape of y is {}'.format(y.shape))

fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
# ax.grid(True)
ax.legend(loc='best')
ax.axis('equal');
```

```
Thin right angle pattern 0.0003047 ms
shape of x_mat_full is (500, 3)
shape of y is (500,)
```

```
/tmp/ipykernel_11446/1400714348.py:29: UserWarning: color is redundantly defined by the 'color' keyword argument and th
e fmt string "ro" (-> color='r'). The keyword argument will take precedence.
  ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
/tmp/ipykernel_11446/1400714348.py:30: UserWarning: color is redundantly defined by the 'color' keyword argument and th
e fmt string "bx" (-> color='b'). The keyword argument will take precedence.
  ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
```




# ANSWER:

ANSWER:

Which patterns was the neural network able to learn quickly and which took longer? What learning rates and numbers of iterations worked well? If you have time, try varying the size of the hidden layer and experiment with different activation functions (e.g. ReLu)

## Solution for: Which patterns was the neural network able to learn quickly and which took longer?

In [51]:

```
def whichPatternTookLonger(indexOfMaxTime):
    switcher = {
        0: "CirclePattern",
        1: "DiamondPattern",
        2: "CenteredSquare",
        3: "ThickRightAnglepattern",
        4: "ThinRightAnglePattern",
    }
    return switcher.get(indexOfMaxTime, "nothing")


minTime = min(timer)
```

```
indexOfMinTime = timer.index(minTime)

pattern = whichPatternTookLonger(indexOfMinTime)
print("Este é o padrão de rede neural que foi capaz de aprender mais rapidamente:", pattern)


maxTime = max(timer)
indexOfMaxTime = timer.index(maxTime)

pattern = whichPatternTookLonger(indexOfMaxTime)
print("Este é o padrão de rede neural que foi capaz de aprender mais demoradamente:", pattern)
```

```
Este é o padrão de rede neural que foi capaz de aprender mais rapidamente: CenteredSquare
Este é o padrão de rede neural que foi capaz de aprender mais demoradamente: ThickRightAnglepattern
```

## Portanto

Este é o padrão de rede neural que foi capaz de aprender mais rapidamente: CenteredSquare Este é o padrão de rede neural que foi capaz de aprender mais demoradamente: ThickRightAnglepattern

Here are some helper functions

## Questão 2

In [66]:
```
def sigmoid(x):
    """
    Sigmoid function
    """
    return 1.0 / (1.0 + np.exp(-x))


def loss_fn(y_true, y_pred, eps=1e-16):
    """
    Loss function we would like to optimize (minimize)
    We are using Logarithmic Loss
    http://scikit-learn.org/stable/modules/model_evaluation.html#log-loss
    """
    y_pred = np.maximum(y_pred,eps)
    y_pred = np.minimum(y_pred,(1-eps))
    return -(np.sum(y_true * np.log(y_pred)) + np.sum((1-y_true)*np.log(1-y_pred)))/len(y_true)
```

```
def forward_pass(W1, W2):
    """
    Does a forward computation of the neural network
    Takes the input `x_mat` (global variable) and produces the output `y_pred`
    Also produces the gradient of the log loss function
    """
    global x_mat
    global y
    global num_
    # First, compute the new predictions `y_pred`
    z_2 = np.dot(x_mat, W_1)
    a_2 = sigmoid(z_2)
    z_3 = np.dot(a_2, W_2)
    y_pred = sigmoid(z_3).reshape((len(x_mat),))
    # Now compute the gradient
    J_z_3_grad = -y + y_pred
    J_W_2_grad = np.dot(J_z_3_grad, a_2)
    a_2_z_2_grad = sigmoid(z_2)*(1-sigmoid(z_2))
    J_W_1_grad = (np.dot((J_z_3_grad).reshape(-1,1), W_2.reshape(-1,1).T)*a_2_z_2_grad).T.dot(x_mat).T
    gradient = (J_W_1_grad, J_W_2_grad)

    # return
    return y_pred, gradient


def plot_loss_accuracy(loss_vals, accuracies):
    fig = plt.figure(figsize=(16, 8))
    fig.suptitle('Log Loss and Accuracy over iterations')

    ax = fig.add_subplot(1, 2, 1)
    ax.plot(loss_vals)
    ax.grid(True)
    ax.set(xlabel='iterations', title='Log Loss')

    ax = fig.add_subplot(1, 2, 2)
    ax.plot(accuracies)
    ax.grid(True)
    ax.set(xlabel='iterations', title='Accuracy');
```

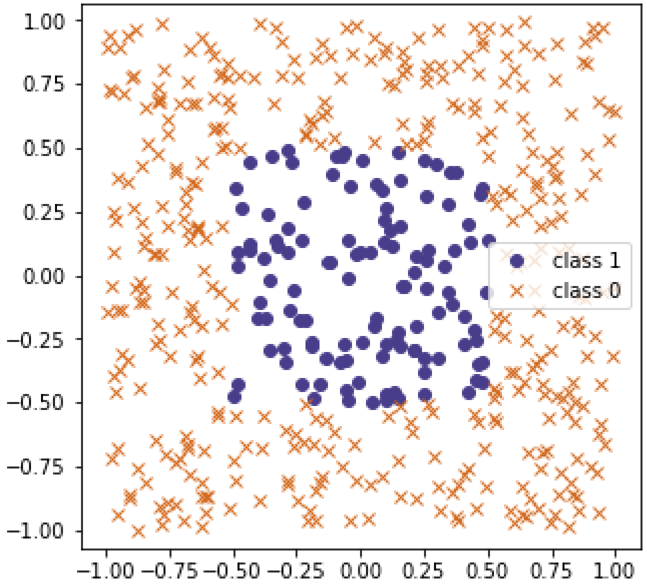
Complete the pseudocode below

## Centered square

In [67]:
```
## This code below generates two x values and a y value according to different patterns
## It also creates a "bias" term (a vector of 1s)
## The goal is then to learn the mapping from x to y using a neural network via back-propagation

import time

num_obs = 500
x_mat_1 = np.random.uniform(-1,1,size = (num_obs,2))
x_mat_bias = np.ones((num_obs,1))
x_mat_full = np.concatenate( (x_mat_1,x_mat_bias), axis=1)


#3 # Centered square
start = time.time()
y = ((np.maximum(np.abs(x_mat_full[:,0]), np.abs(x_mat_full[:,1])))<.5).astype(int)
end = time.time()

timeCenteredSquare = round(end - start,7)
timer.append(timeCenteredSquare)

print("Centered square",timeCenteredSquare,"ms")

print('shape of x_mat_full is {}'.format(x_mat_full.shape))
print('shape of y is {}'.format(y.shape))

fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
# ax.grid(True)
ax.legend(loc='best')
ax.axis('equal');
```

```
Centered square 0.0002558 ms
shape of x_mat_full is (500, 3)
shape of y is (500,)
```

```
/tmp/ipykernel_11446/3138982794.py:27: UserWarning: color is redundantly defined by the 'color' keyword argument and th
e fmt string "ro" (-> color='r'). The keyword argument will take precedence.
  ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
/tmp/ipykernel_11446/3138982794.py:28: UserWarning: color is redundantly defined by the 'color' keyword argument and th
e fmt string "bx" (-> color='b'). The keyword argument will take precedence.
  ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
```

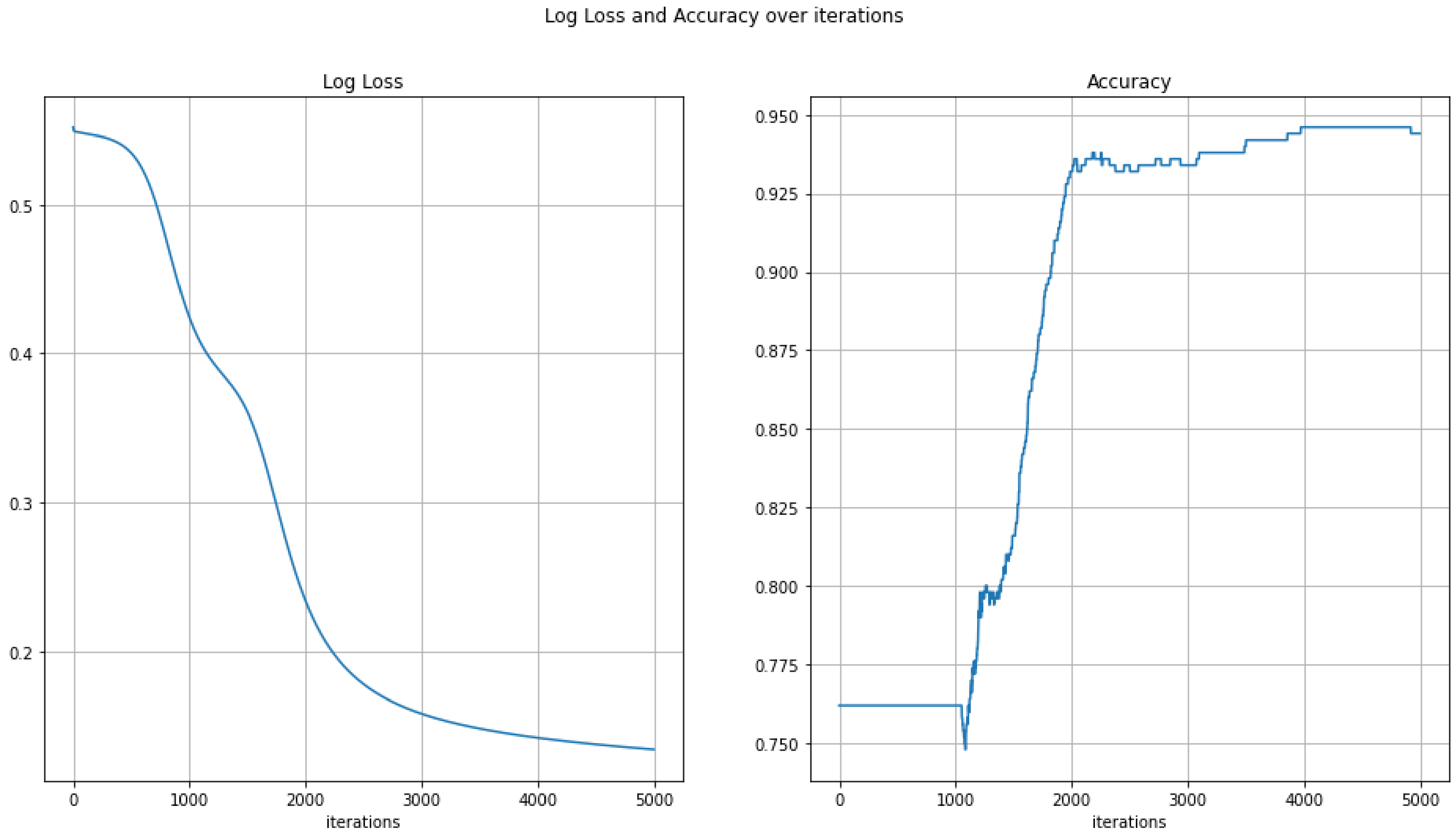



In [69]:

```
#### Initialize the network parameters

np.random.seed(1241) #semente para numero aleatorio

#pesos
W_1 = np.random.uniform(-1,1,size=(3,4))
W_2 = np.random.uniform(-1,1,size=(4))
num_iter = 5000
learning_rate = .001
x_mat = x_mat_full


loss_vals, accuracies = [], []
for i in range(num_iter):
    ### Do a forward computation, and get the gradient and y predicted
    y_pred, (J_W_1_grad, J_W_2_grad) = forward_pass(W_1, W_2)

    ## Update the weight matrices,a step in the opposite direction
    W_1 = W_1 - learning_rate*J_W_1_grad
    W_2 = W_2 - learning_rate*J_W_2_grad

    ### Compute the loss and accuracy
    curr_loss = loss_fn(y,y_pred) #Error function
```

```
    loss_vals.append(curr_loss)
    acc = np.sum((y_pred>=.5) == y)/num_obs
    accuracies.append(acc)

    ## Print the loss and accuracy for every 200th iteration
    if((i%200) == 0):
        print('iteration {}, log loss is {:.4f}, accuracy is {}'.format(
            i, curr_loss, acc
        ))
plot_loss_accuracy(loss_vals, accuracies)
```

```
iteration 0, log loss is 0.5518, accuracy is 0.762
iteration 200, log loss is 0.5463, accuracy is 0.762
iteration 400, log loss is 0.5409, accuracy is 0.762
iteration 600, log loss is 0.5234, accuracy is 0.762
iteration 800, log loss is 0.4774, accuracy is 0.762
iteration 1000, log loss is 0.4240, accuracy is 0.762
iteration 1200, log loss is 0.3941, accuracy is 0.792
iteration 1400, log loss is 0.3744, accuracy is 0.802
iteration 1600, log loss is 0.3402, accuracy is 0.846
iteration 1800, log loss is 0.2835, accuracy is 0.896
iteration 2000, log loss is 0.2339, accuracy is 0.932
iteration 2200, log loss is 0.2031, accuracy is 0.936
iteration 2400, log loss is 0.1845, accuracy is 0.932
iteration 2600, log loss is 0.1725, accuracy is 0.934
iteration 2800, log loss is 0.1642, accuracy is 0.934
iteration 3000, log loss is 0.1581, accuracy is 0.934
iteration 3200, log loss is 0.1534, accuracy is 0.938
iteration 3400, log loss is 0.1497, accuracy is 0.938
iteration 3600, log loss is 0.1467, accuracy is 0.942
iteration 3800, log loss is 0.1441, accuracy is 0.942
iteration 4000, log loss is 0.1419, accuracy is 0.946
iteration 4200, log loss is 0.1400, accuracy is 0.946
iteration 4400, log loss is 0.1383, accuracy is 0.946
iteration 4600, log loss is 0.1368, accuracy is 0.946
iteration 4800, log loss is 0.1354, accuracy is 0.946
```




O backpropagation nos diz como fazer um único ajuste usando o cálculo, ao comparar a saída com as respostas corretas e computar a função de perda, realizando o ajuste e repetir, por iterações a analisar graficamente log loss e accuracy para coletar informações a respeito de métricas do modelo.

Log Loss é utilizado para informar quão perto a predição de probabilidade está correspondendo com a realidade, em outras palavras a performance, com seus valores limitados entre 0 e 1. Pelo gráfico acima, perceb-se que após a correção dos parâmetos, a partir da iteração número 3000, que o valor o valor do Log Loss se altera a uma taxa menor, a um valor baixo, o ideal.

Sendo a acurácia uma medida de quão o modelo predisse corretamente em relação a todas as predições, podemos inferir, a partir do gráfico que a partir da iteração de número 2000 a acurácia começa a atingir valores melhores para o modelo. A partir do gráfico de log loss, consideramos a iteração de valor 4000 a mais interessante para o modelo, como um todo, por estar em uma região com log loss baixo, e estabilizada em termos de décadas logaritmicas, e acurácia elevada e estabilizada.

In [ ]:

```

```