Hands_on_Activity_6_1_Neural_Networks_DONE

April 2, 2024

1 Activity 1.1 : Neural Networks

Technological Institute of the Philippines	Quezon City - Computer Engineering
Course Code:	CPE 019
Code Title:	Emerging Technologies in CpE 2 - Fundamentals of Computer Vision
2nd Semester	AY 2023-2024

|

ACTIVITY NO. | **Hands-on Activity 6.1 - Neural Networks** **Name** | Dela Cruz, Irish **Section** | CPE32S3 **Date Performed**: | 03/27/2024 **Date Submitted**: | 04/02/2024 **Instructor**: | Engr. Roman M. Richard

Objective(s): This activity aims to demonstrate the concepts of neural networks

Intended Learning Outcomes (ILOs):

- Demonstrate how to use activation function in neural networks
- Demonstrate how to apply feedforward and backpropagation in neural networks

Resources:

- Jupyter Notebook

Procedure: Import the libraries

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Define and plot an activation function

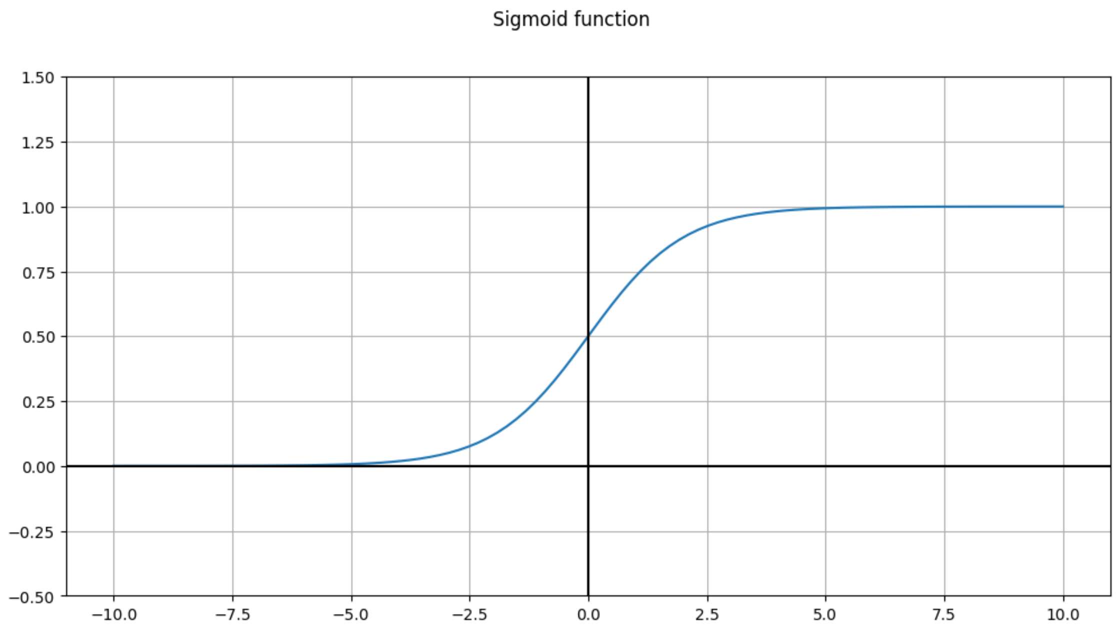
1.0.1 Sigmoid function:

$$\sigma = \frac{1}{1+e^{-x}}$$

σ ranges from (0, 1). When the input x is negative, σ is close to 0. When x is positive, σ is close to 1. At $x = 0$, $\sigma = 0.5$

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

[ ]:
```
# Plot the sigmoid function
vals = np.linspace(-10, 10, num=100, dtype=np.float32)
activation = sigmoid(vals)
fig = plt.figure(figsize=(12,6))
fig.suptitle('Sigmoid function')
plt.plot(vals, activation)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.yticks()
plt.ylim([-0.5, 1.5]);
```

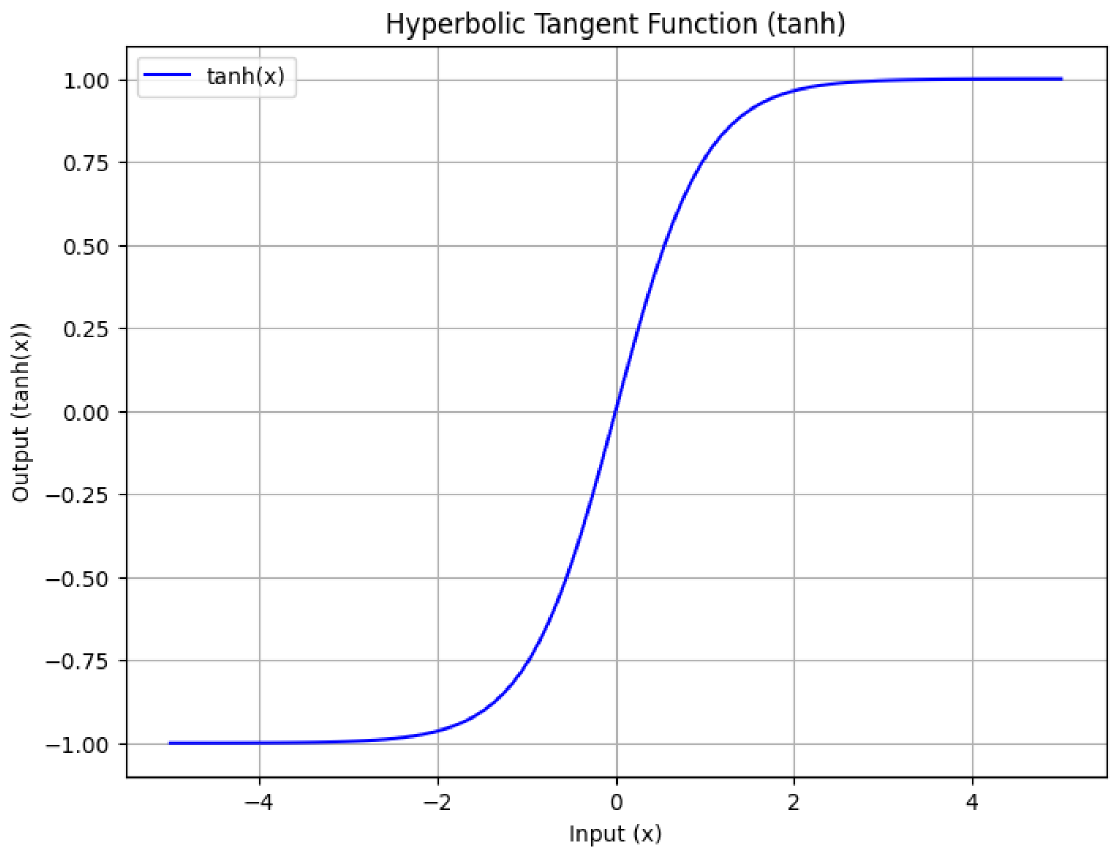


Choose any activation function and create a method to define that function.

[11]:
```
#type your code here
import numpy as np

def tanh(x):
    """Hyperbolic Tangent Function"""
    return np.tanh(x)
```

Plot the activation function

[12]:
```
#type your code here
x_values = np.linspace(-5, 5, 100)
y_values = tanh(x_values)

plt.figure(figsize=(8, 6))
plt.plot(x_values, y_values, label='tanh(x)', color='blue')
plt.title('Hyperbolic Tangent Function (tanh)')
plt.xlabel('Input (x)')
plt.ylabel('Output (tanh(x))')
plt.grid(True)
plt.legend()
plt.show()
```



1.0.2 Neurons as boolean logic gates

1.0.3 OR Gate

OR gate truth table

Input

Output

0

0

0

0

1

1

1

0

1

1

1

1

A neuron that uses the sigmoid activation function outputs a value between (0, 1). This naturally leads us to think about boolean values.

By limiting the inputs of x_1 and x_2 to be in $\{0, 1\}$, we can simulate the effect of logic gates with our neuron. The goal is to find the weights , such that it returns an output close to 0 or 1 depending on the inputs.

What numbers for the weights would we need to fill in for this gate to output OR logic? Observe from the plot above that $\sigma(z)$ is close to 0 when z is largely negative (around -10 or less), and is close to 1 when z is largely positive (around +10 or greater).

$$z = w_1 x_1 + w_2 x_2 + b$$

Let's think this through:

- When x_1 and x_2 are both 0, the only value affecting z is b. Because we want the result for (0, 0) to be close to zero, b should be negative (at least -10)
- If either x_1 or x_2 is 1, we want the output to be close to 1. That means the weights associated with x_1 and x_2 should be enough to offset b to the point of causing z to be at least 10.
- Let's give b a value of -10. How big do we need w_1 and w_2 to be?
  - At least +20
- So let's try out $w_1 = 20$, $w_2 = 20$, and $b = -10$!

```
[ ]: def logic_gate(w1, w2, b):
         # Helper to create logic gate functions
         # Plug in values for weight_a, weight_b, and bias
         return lambda x1, x2: sigmoid(w1 * x1 + w2 * x2 + b)

     def test(gate):
         # Helper function to test out our weight functions.
         for a, b in (0, 0), (0, 1), (1, 0), (1, 1):
             print("{}, {}: {}".format(a, b, np.round(gate(a, b))))
```

```
[ ]: or_gate = logic_gate(20, 20, -10)
     test(or_gate)
```

```
0, 0: 0.0
0, 1: 1.0
1, 0: 1.0
1, 1: 1.0
```

OR gate truth table

Input

Output

0

0

0

0

1

1

1

0

1

1

1

1

Try finding the appropriate weight values for each truth table.

1.0.4 AND Gate

AND gate truth table

Input

Output

```
0

0

0

0

1

0

1

0

0

1

1

1
```

Try to figure out what values for the neurons would make this function as an AND gate.

```
[ ]: # Fill in the w1, w2, and b parameters such that the truth table matches
w1 = 1
w2 = 1
b = -1.5
and_gate = logic_gate(w1, w2, b)

print("AND Gate:")
test(and_gate)
```

```
AND Gate:
0, 0: 0.0
0, 1: 0.0
1, 0: 0.0
1, 1: 1.0
```

Do the same for the NOR gate and the NAND gate.

```
[ ]: #NOR Gate
w1 = -20
w2 = -20
b = 10
nor_gate = logic_gate(w1, w2, b)

print("NOR Gate:")
test(nor_gate)
```

```
NOR Gate:
0, 0: 1.0
0, 1: 0.0
```

```
1, 0: 0.0
1, 1: 0.0
```

[ ]:
```python
#NAND Gate
w1 = -0.5
w2 = -0.5
b = 0.7
nand_gate = logic_gate(w1, w2, b)

print("NAND Gate:")
test(nand_gate)
```

```
NAND Gate:
0, 0: 1.0
0, 1: 1.0
1, 0: 1.0
1, 1: 0.0
```

1.1 Limitation of single neuron

Here's the truth table for XOR:

1.1.1 XOR (Exclusive Or) Gate

XOR gate truth table

Input

Output

0

0

0

0

1

1

1

0

1

1

1

0

Now the question is, can you create a set of weights such that a single neuron can output this property?

It turns out that you cannot. Single neurons can't correlate inputs, so it's just confused. So individual neurons are out. Can we still use neurons to somehow form an XOR gate?

[ ]:
```
# Make sure you have or_gate, nand_gate, and and_gate working from above!
def xor_gate(a, b):
    c = or_gate(a, b)
    d = nand_gate(a, b)
    return and_gate(c, d)
test(xor_gate)
```

```
0, 0: 0.0
0, 1: 1.0
1, 0: 1.0
1, 1: 0.0
```

1.2 Feedforward Networks

The feed-forward computation of a neural network can be thought of as matrix calculations and activation functions. We will do some actual computations with matrices to see this in action.

1.3 Exercise

Provided below are the following:

- Three weight matrices `W_1`, `W_2` and `W_3` representing the weights in each layer. The convention for these matrices is that each $W_{i,j}$ gives the weight from neuron i in the previous (left) layer to neuron j in the next (right) layer.

- A vector `x_in` representing a single input and a matrix `x_mat_in` representing 7 different inputs.
- Two functions: `soft_max_vec` and `soft_max_mat` which apply the soft_max function to a single vector, and row-wise to a matrix.

The goals for this exercise are: 1. For input `x_in` calculate the inputs and outputs to each layer (assuming sigmoid activations for the middle two layers and soft_max output for the final layer. 2. Write a function that does the entire neural network calculation for a single input 3. Write a function that does the entire neural network calculation for a matrix of inputs, where each row is a single input. 4. Test your functions on `x_in` and `x_mat_in`.

This illustrates what happens in a NN during one single forward pass. Roughly speaking, after this forward pass, it remains to compare the output of the network to the known truth values, compute the gradient of the loss function and adjust the weight matrices `W_1`, `W_2` and `W_3` accordingly, and iterate. Hopefully this process will result in better weight matrices and our loss will be smaller afterwards

[75]:
```
W_1 = np.array([[2,-1,1,4],[-1,2,-3,1],[3,-2,-1,5]])
W_2 = np.array([[3,1,-2,1],[-2,4,1,-4],[-1,-3,2,-5],[3,1,1,1]])
W_3 = np.array([[-1,3,-2],[1,-1,-3],[3,-2,2],[1,2,1]])
x_in = np.array([.5,.8,.2])
```

```python
x_mat_in = np.array([[.5,.8,.2],[.1,.9,.6],[.2,.2,.3],[.6,.1,.9],[.5,.5,.4],[.
↪9,.1,.9],[.1,.8,.7]])

def soft_max_vec(vec):
    return np.exp(vec)/(np.sum(np.exp(vec)))

def soft_max_mat(mat):
    return np.exp(mat)/(np.sum(np.exp(mat),axis=1).reshape(-1,1))

print('the matrix W_1\n')
print(W_1)
print('-'*30)
print('vector input x_in\n')
print(x_in)
print ('-'*30)
print('matrix input x_mat_in -- starts with the vector `x_in`\n')
print(x_mat_in)
```

```
the matrix W_1

[[ 2 -1  1  4]
 [-1  2 -3  1]
 [ 3 -2 -1  5]]
------------------------------
vector input x_in

[0.5 0.8 0.2]
------------------------------
matrix input x_mat_in -- starts with the vector `x_in`

[[0.5 0.8 0.2]
 [0.1 0.9 0.6]
 [0.2 0.2 0.3]
 [0.6 0.1 0.9]
 [0.5 0.5 0.4]
 [0.9 0.1 0.9]
 [0.1 0.8 0.7]]
```

1.4 Exercise

1. Get the product of array x_in and W_1 (z2)
2. Apply sigmoid function to z2 that results to a2
3. Get the product of a2 and z2 (z3)
4. Apply sigmoid function to z3 that results to a3
5. Get the product of a3 and z3 that results to z4

[76]:
```
#type your code here
#1
z2 = np.dot(x_in, W_1)
print("Result: ", z2)
```

```
Result:  [ 0.8  0.7 -2.1  3.8]
```

[77]:
```
#2
a2 = sigmoid(z2)
print("Result (a2): ", a2)
```

```
Result (a2):  [0.68997448 0.66818777 0.10909682 0.97811873]
```

[78]:
```
#3
z3 = a2 *z2
print("Result (z3): ", z3)
```

```
Result (z3):  [ 0.55197958  0.46773144 -0.22910332  3.71685117]
```

[79]:
```
#4
a3 = sigmoid(z3)
print("Result (a3): ", a3)
```

```
Result (a3):  [0.63459475 0.61484668 0.44297339 0.97626657]
```

[80]:
```
#5
z4 = a3 * z3
print("Result (z4): ", z4)
```

```
Result (z4):  [ 0.35028335  0.28758312 -0.10148668  3.62863755]
```

[81]:
```
def soft_max_vec(vec):
    return np.exp(vec)/(np.sum(np.exp(vec)))

def soft_max_mat(mat):
    return np.exp(mat)/(np.sum(np.exp(mat),axis=1).reshape(-1,1))
```

7. Apply soft_max_vec function to z4 that results to y_out

[83]:
```
#type your code here
y_out = soft_max_vec(z4)
print("Result (y_out): ", y_out)
```

```
Result (y_out):  [0.03435506 0.03226713 0.02186701 0.9115108 ]
```

[84]:
```
## A one-line function to do the entire neural net computation

def nn_comp_vec(x):
    return soft_max_vec(sigmoid(sigmoid(np.dot(x,W_1)).dot(W_2)).dot(W_3))

def nn_comp_mat(x):
    return soft_max_mat(sigmoid(sigmoid(np.dot(x,W_1)).dot(W_2)).dot(W_3))
```

[85]:
```
nn_comp_vec(x_in)
```

[85]:
```
array([0.72780576, 0.26927918, 0.00291506])
```

[86]:
```
nn_comp_mat(x_mat_in)
```

[86]:
```
array([[0.72780576, 0.26927918, 0.00291506],
       [0.62054212, 0.37682531, 0.00263257],
       [0.69267581, 0.30361576, 0.00370844],
       [0.36618794, 0.63016955, 0.00364252],
       [0.57199769, 0.4251982 , 0.00280411],
       [0.38373781, 0.61163804, 0.00462415],
       [0.52510443, 0.4725011 , 0.00239447]])
```

1.5 Backpropagation

The backpropagation in this part will be used to train a multi-layer perceptron (with a single hidden layer). Different patterns will be used and the demonstration on how the weights will converge. The different parameters such as learning rate, number of iterations, and number of data points will be demonstrated

[87]:
```
#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? I think the neural network that able to learn quickly are Circle (only calculation of Eucliden distance to origin) and Centered pattern (comparing absolute values of coordinates) since neural network performs well with the smooth and simple pattern. And the rest took longer since they have complex pattern.

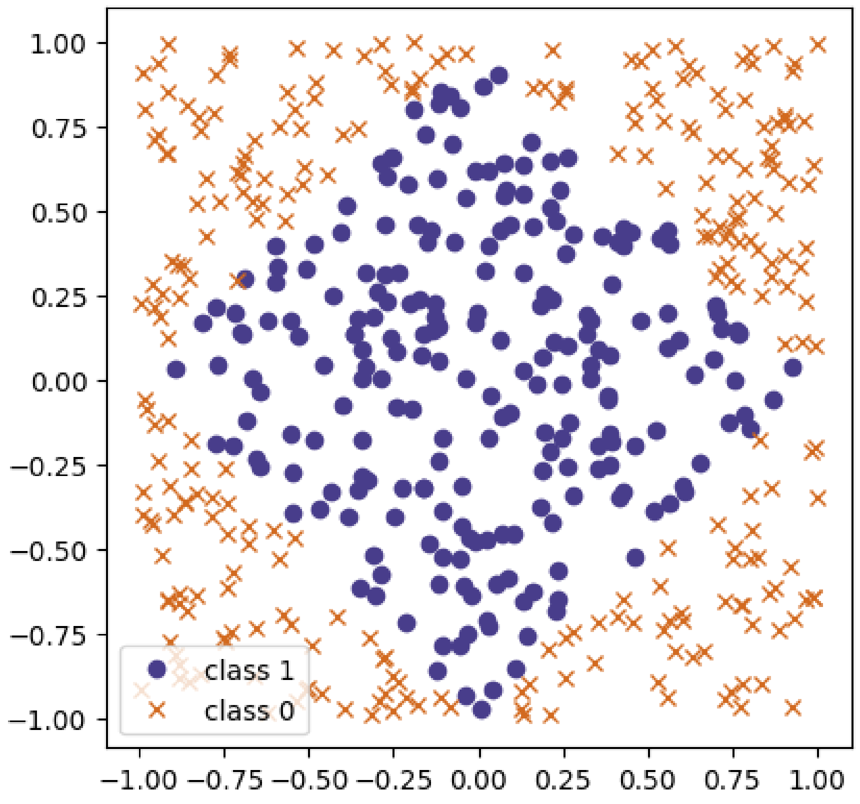
- What learning rates and numbers of iterations worked well? Moderate learning rate start as 0.01 or 0.001 as often good starting point for optimization problems. If the case that the training seems so slow adjust the learning rate. Number of Iterations ususally start at 100 so it gives a better presentation since limited to only 100 process.

[88]:
```
## 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

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.

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

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

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

# # Thick Right Angle pattern
# y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.
↪maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>-.5)).astype(int)

# # Thin right angle pattern
# y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.
↪maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>0)).astype(int)


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');
```

```
shape of x_mat_full is (500, 3)
shape of y is (500,)
```

```
<ipython-input-88-0f8bccfdade3>: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')
<ipython-input-88-0f8bccfdade3>: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')
```



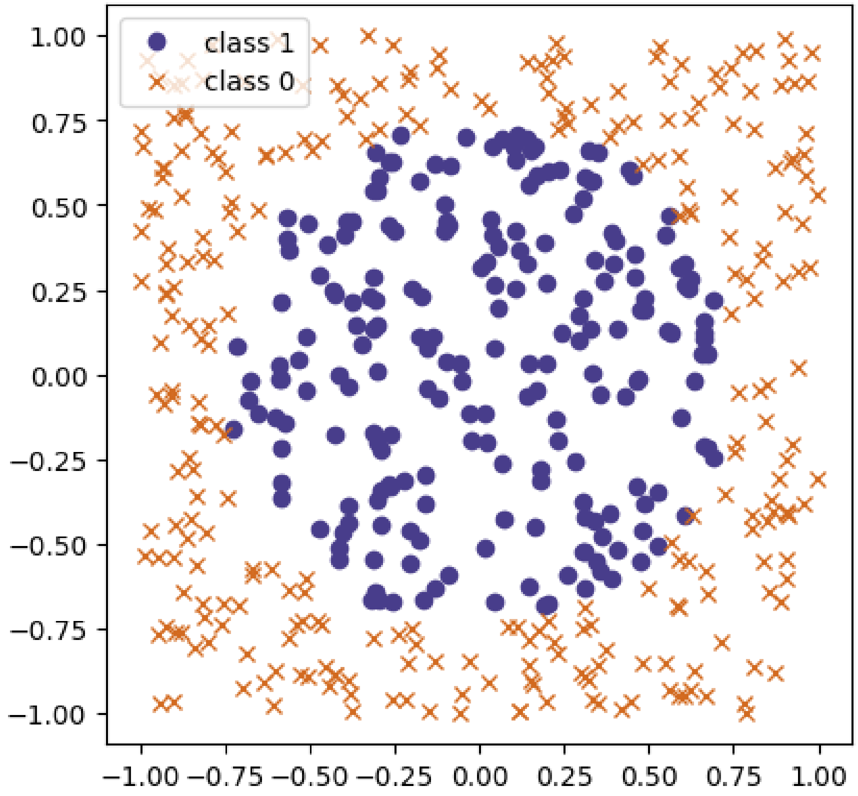
[30]:
```
## This code below generates two x values and a y value according to different␣
↪patterns
## It also creates a "bias" term (a vector of 1s)
```

```python
## The goal is then to learn the mapping from x to y using a neural network via␣
 ↪back-propagation

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.

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

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

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

# # Thick Right Angle pattern
# y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.
 ↪maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>-.5)).astype(int)

# # Thin right angle pattern
# y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.
 ↪maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>0)).astype(int)


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');
```

```
shape of x_mat_full is (500, 3)
shape of y is (500,)
```

```
<ipython-input-30-1d370139ed12>:32: UserWarning: color is redundantly defined by
the 'color' keyword argument and the fmt string "ro" (-> color='r'). The keyword
argument will take precedence.
```

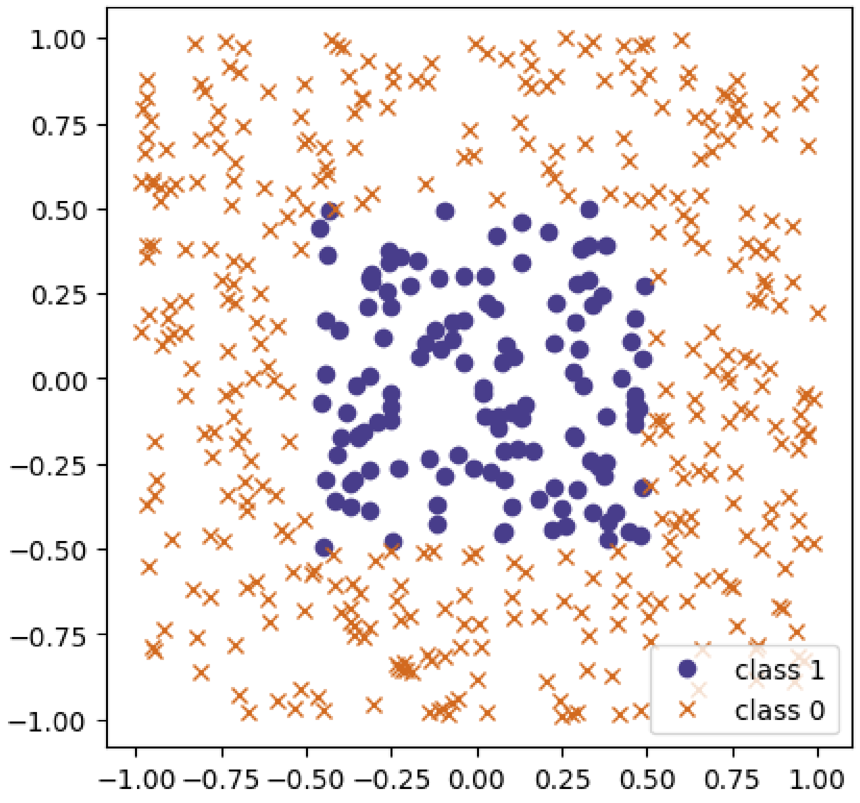
```
  ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1',
color='darkslateblue')
<ipython-input-30-1d370139ed12>:33: UserWarning: color is redundantly defined by
the 'color' keyword argument and the fmt string "bx" (-> color='b'). The keyword
argument will take precedence.
  ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0',
color='chocolate')
```



[32]:
```
## 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

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

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

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

# # Thick Right Angle pattern
# y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.
 ↪maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>-.5)).astype(int)

# # Thin right angle pattern
# y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.
 ↪maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>0)).astype(int)


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');
```

```
shape of x_mat_full is (500, 3)
shape of y is (500,)
```

```
<ipython-input-32-22d52f40b73a>: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')
<ipython-input-32-22d52f40b73a>: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')
```

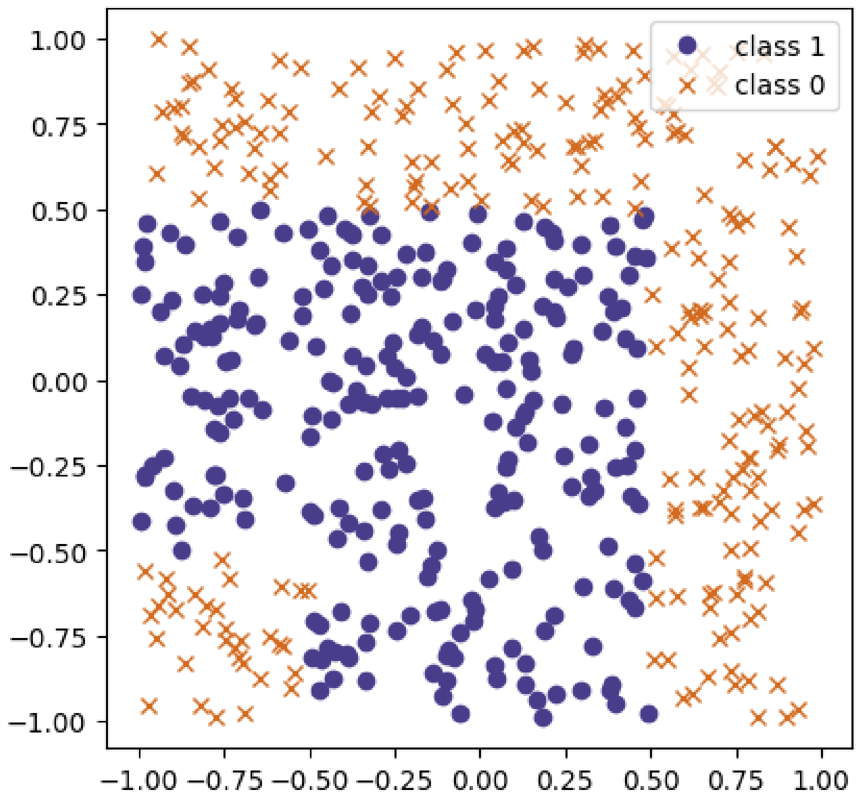


[33]:
```
## This code below generates two x values and a y value according to different␣
↪patterns
## It also creates a "bias" term (a vector of 1s)
## The goal is then to learn the mapping from x to y using a neural network via␣
↪back-propagation

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.

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

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

# # Centered square
```

```python
#y = ((np.maximum(np.abs(x_mat_full[:,0]), np.abs(x_mat_full[:,1])))<.5).
 ↪astype(int)

# # Thick Right Angle pattern
y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.
 ↪maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>-.5)).astype(int)

# # Thin right angle pattern
# y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.
 ↪maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>0)).astype(int)



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');
```

```
shape of x_mat_full is (500, 3)
shape of y is (500,)
```

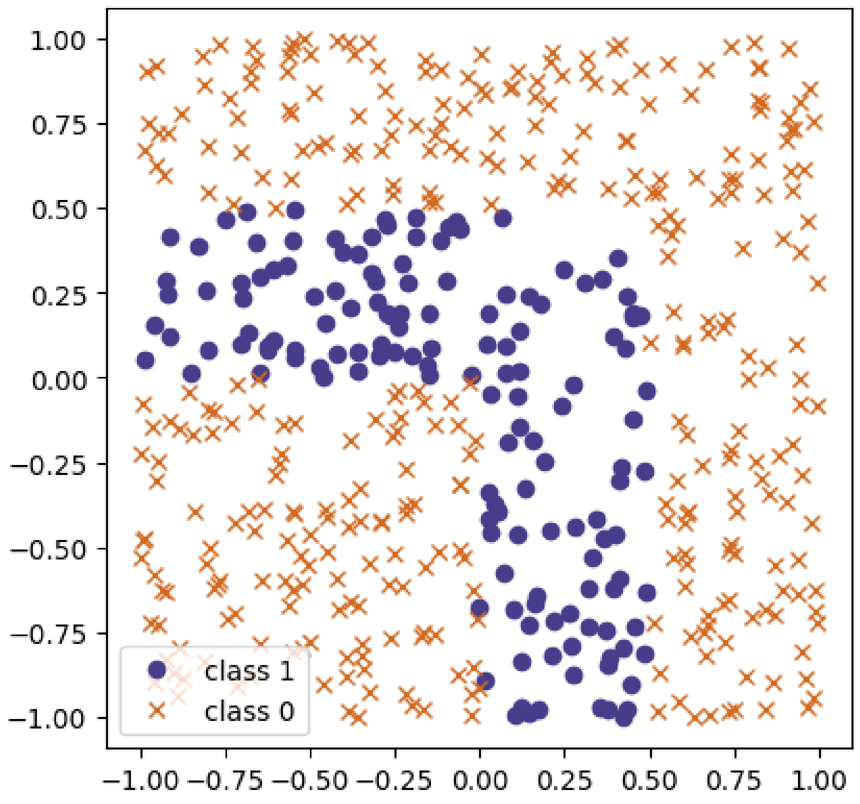
```
<ipython-input-33-51767655d605>:32: UserWarning: color is redundantly defined by
the 'color' keyword argument and the fmt string "ro" (-> color='r'). The keyword
argument will take precedence.
  ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1',
color='darkslateblue')
<ipython-input-33-51767655d605>:33: 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')
```



[34]:
```
## 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

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.

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

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

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

# # Thick Right Angle pattern
# y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.
↪maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>-.5)).astype(int)

# # Thin right angle pattern
y = (((np.maximum((x_mat_full[:,0]), (x_mat_full[:,1])))<.5) & ((np.
↪maximum((x_mat_full[:,0]), (x_mat_full[:,1])))>0)).astype(int)


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');
```

```
shape of x_mat_full is (500, 3)
shape of y is (500,)
```

```
<ipython-input-34-fa2c7fe8ce53>: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')
<ipython-input-34-fa2c7fe8ce53>: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')
```



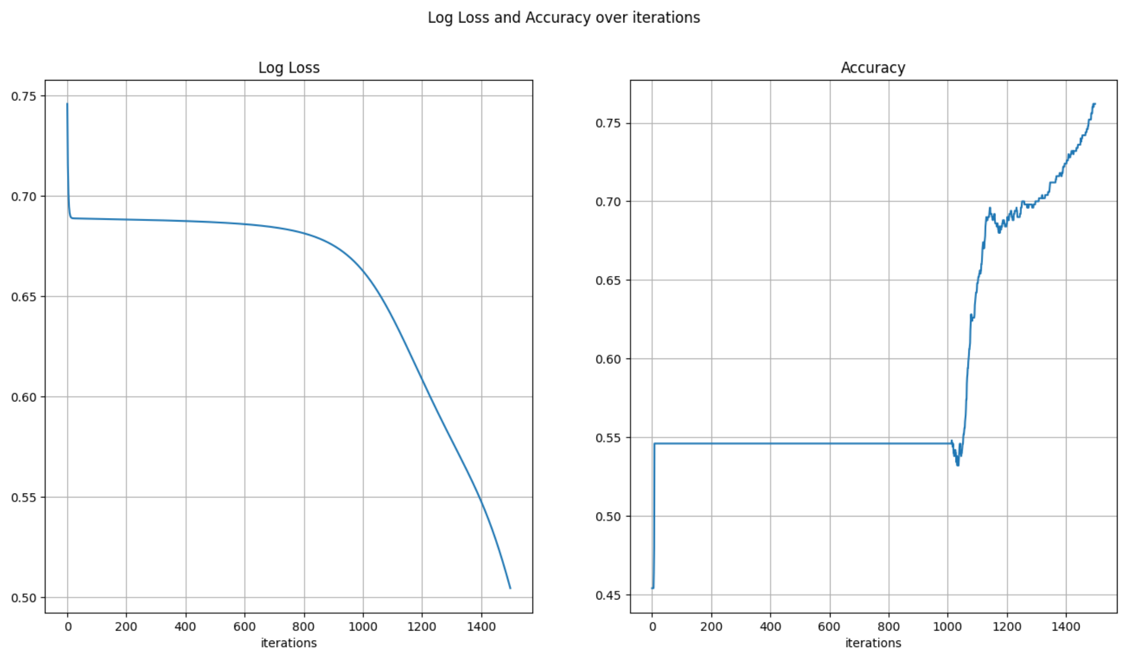
[89]:
```python
def sigmoid(x):
    """
    Sigmoid function
    """
    return 1.0 / (1.0 + np.exp(-x))


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

[102]:
```
#### Initialize the network parameters

np.random.seed(1234)

# Assuming x_mat has 3 features
W_1 = np.random.uniform(-1, 1, size=(3, 4))
W_2 = np.random.uniform(-1, 1, size=(4))
```

```python
num_iter = 1500

learning_rate = 0.001
x_mat = x_mat_full

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

    # Update the weight matrices
    W_1 -= learning_rate * J_W_1_grad
    W_2 -= learning_rate * J_W_2_grad

    # Compute the loss and accuracy
    Loss = loss_fn(y, y_pred)
    loss_vals.append(Loss)

    Accuracy = np.sum((y_pred >= 0.5) == y) / num_obs
    accuracies.append(Accuracy)

    # Print the loss and accuracy for every 200th iteration
    if i % 200 == 0:
        print(f"Iteration {i}: Loss {Loss:.4f}, Accuracy {Accuracy:.4f}")

plot_loss_accuracy(loss_vals, accuracies)
```

```
Iteration 0: Loss 0.7460, Accuracy 0.4540
Iteration 200: Loss 0.6883, Accuracy 0.5460
Iteration 400: Loss 0.6875, Accuracy 0.5460
Iteration 600: Loss 0.6860, Accuracy 0.5460
Iteration 800: Loss 0.6814, Accuracy 0.5460
Iteration 1000: Loss 0.6627, Accuracy 0.5460
Iteration 1200: Loss 0.6087, Accuracy 0.6860
Iteration 1400: Loss 0.5479, Accuracy 0.7240
```

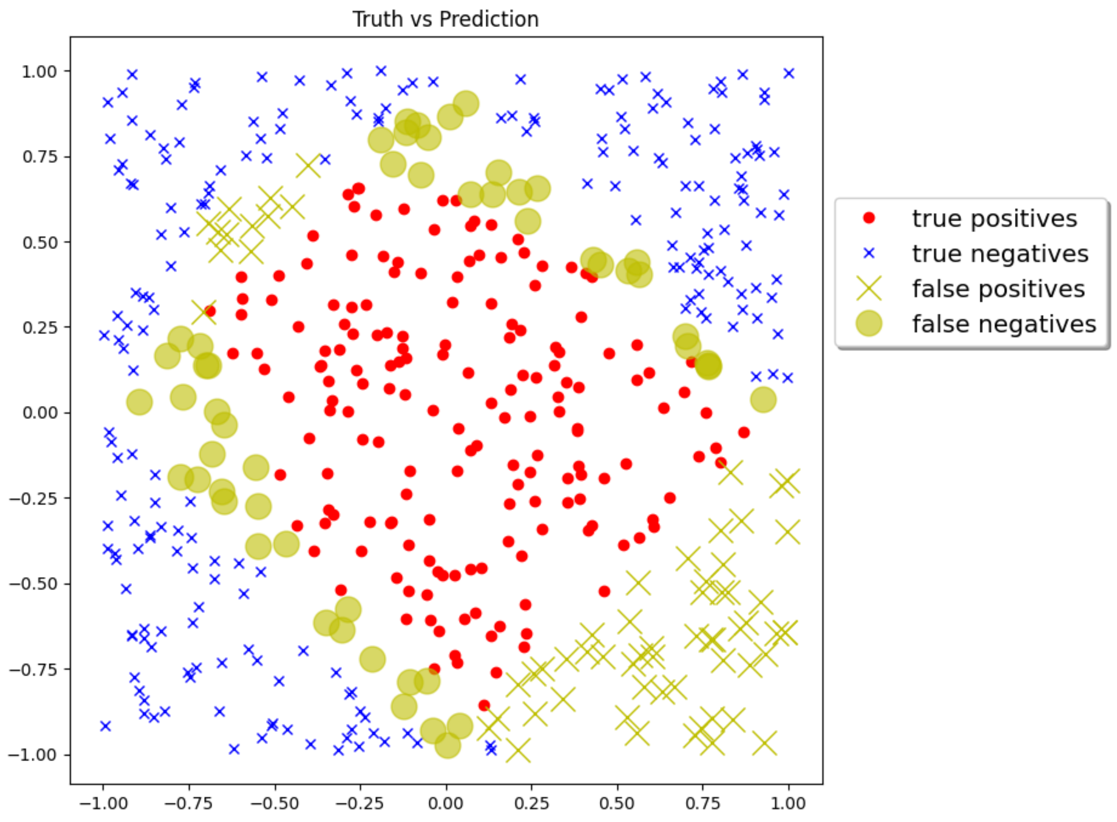


2 Analysis

As the no. of iteration increases, the log loss decreases, which indicates that the model was imporving. The lowest log is around 0.5 which suggest that the model predicts the probabilities more accurately at the end of iterations. The accuracy starts at 0.5 and increase gradually, reaching up to the 0.75. After the peak, the accuracy fluctuates slightly, but remains relatively stable. The model is consistent and making more accurate prediction

Plot the predicted answers, with mistakes in yellow

[103]:
```
pred1 = (y_pred>=.5)
pred0 = (y_pred<.5)

fig, ax = plt.subplots(figsize=(8, 8))
# true predictions
ax.plot(x_mat[pred1 & (y==1),0],x_mat[pred1 & (y==1),1], 'ro', label='true␣
↪positives')
ax.plot(x_mat[pred0 & (y==0),0],x_mat[pred0 & (y==0),1], 'bx', label='true␣
↪negatives')
# false predictions
ax.plot(x_mat[pred1 & (y==0),0],x_mat[pred1 & (y==0),1], 'yx', label='false␣
↪positives', markersize=15)
ax.plot(x_mat[pred0 & (y==1),0],x_mat[pred0 & (y==1),1], 'yo', label='false␣
↪negatives', markersize=15, alpha=.6)
ax.set(title='Truth vs Prediction')
ax.legend(bbox_to_anchor=(1, 0.8), fancybox=True, shadow=True,␣
 ↪fontsize='x-large');
```



3 Remarks

TP - correctly predicted positive values - 20% TN - correctly predicted negative - 50% FP - incorrectly predicted positive values - 10% FN - incorrectly predicted negative values - 15%

Assuming that most of the instances here is from True Negative which means there's a lot of correctly predicted on negative values. The pattern mostly distinguish the negative values. Then next one, who also gets more instances is from True positive which predicted a 20% positive values.

Supplementary Activity

1. Use a different weights , input and activation function
2. Apply feedforward and backpropagation
3. Plot the loss and accuracy for every 300th iteration

Conclusion #type your conclusion here

This activity gives me an insight for the activation that can be used for neural networks. Also, how the movement of information flow in neural networks and the difference between the 2. Activation functions introduce non-linearity, allowing neural networks to capture intricate relationships within data. During feedforward, inputs traverse through the network, producing predictions. Backpropagation refines the network by adjusting weights in response to prediction errors, enhancing performance. These mechanisms are pivotal in training neural networks, empowering them to glean insights from data effectively.

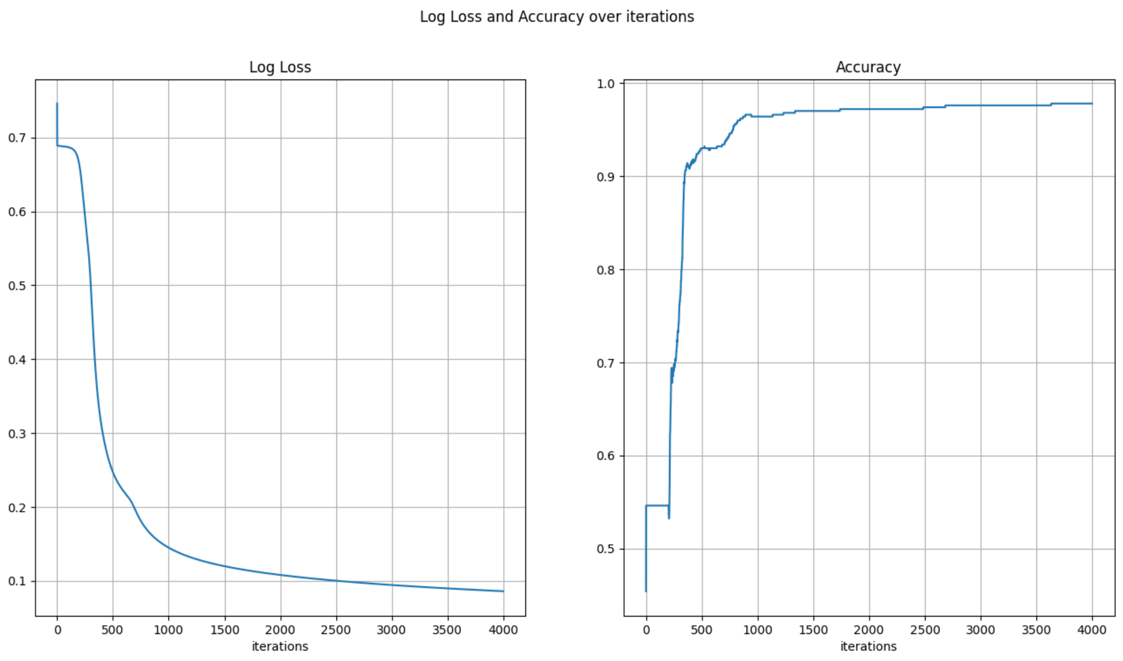
[104]:
```
def tanh(x):
    """Hyperbolic Tangent Function"""
    return np.tanh(x)

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
```

```python
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');
```

[115]:
```python
np.random.seed(1234)

# Assuming x_mat has 3 features
W_1 = np.random.uniform(-1, 1, size=(3, 4))
W_2 = np.random.uniform(-1, 1, size=(4))
num_iter = 4000

learning_rate = 0.005
x_mat = x_mat_full

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

    # Update the weight matrices
    W_1 -= learning_rate * J_W_1_grad
    W_2 -= learning_rate * J_W_2_grad

    # Compute the loss and accuracy
    Loss = loss_fn(y, y_pred)
    loss_vals.append(Loss)

    Accuracy = np.sum((y_pred >= 0.5) == y) / num_obs
    accuracies.append(Accuracy)

    # Print the loss and accuracy for every 200th iteration
    if i % 200 == 0:
        print(f"Iteration {i}: Loss {Loss:.4f}, Accuracy {Accuracy:.4f}")
```

```
Iteration 0: Loss 0.7460, Accuracy 0.4540
Iteration 200: Loss 0.6621, Accuracy 0.5460
Iteration 400: Loss 0.3072, Accuracy 0.9120
Iteration 600: Loss 0.2202, Accuracy 0.9300
Iteration 800: Loss 0.1698, Accuracy 0.9560
Iteration 1000: Loss 0.1446, Accuracy 0.9640
Iteration 1200: Loss 0.1314, Accuracy 0.9660
Iteration 1400: Loss 0.1228, Accuracy 0.9700
Iteration 1600: Loss 0.1166, Accuracy 0.9700
Iteration 1800: Loss 0.1117, Accuracy 0.9720
Iteration 2000: Loss 0.1076, Accuracy 0.9720
Iteration 2200: Loss 0.1042, Accuracy 0.9720
Iteration 2400: Loss 0.1012, Accuracy 0.9720
Iteration 2600: Loss 0.0985, Accuracy 0.9740
Iteration 2800: Loss 0.0962, Accuracy 0.9760
Iteration 3000: Loss 0.0940, Accuracy 0.9760
Iteration 3200: Loss 0.0921, Accuracy 0.9760
Iteration 3400: Loss 0.0903, Accuracy 0.9760
Iteration 3600: Loss 0.0886, Accuracy 0.9760
Iteration 3800: Loss 0.0870, Accuracy 0.9780
```

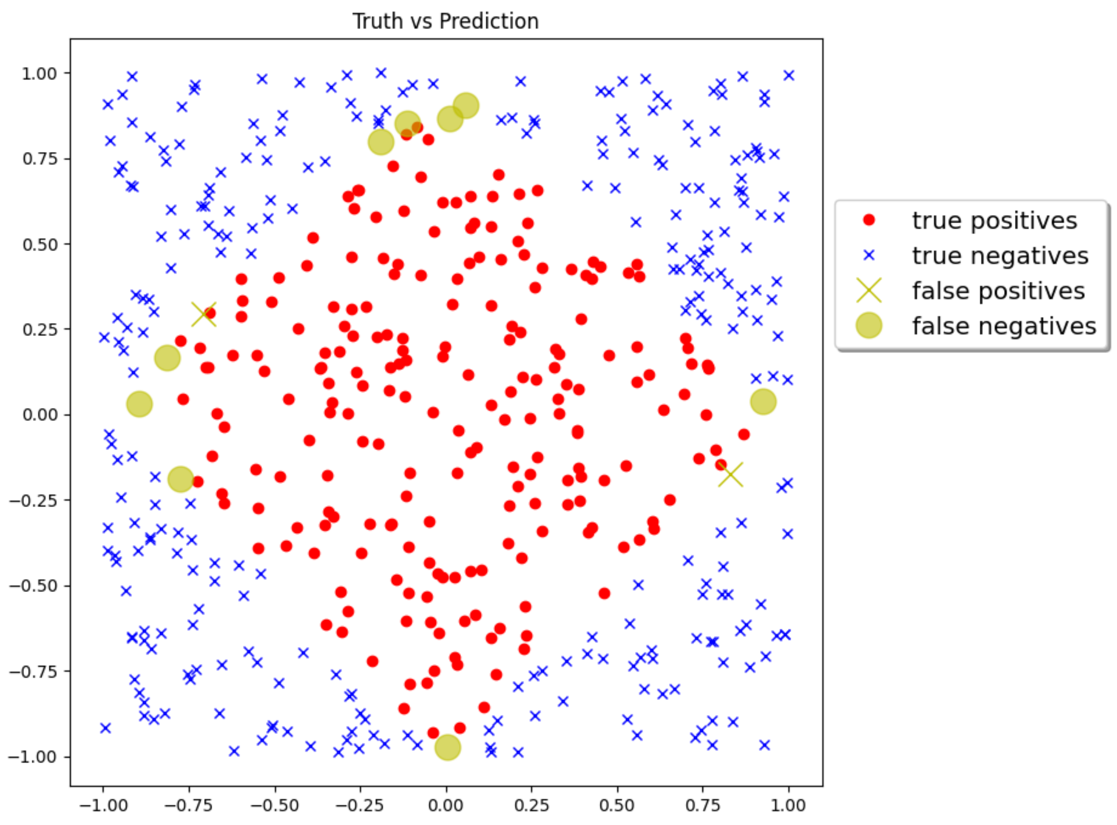
[116]:
```
plot_loss_accuracy(loss_vals, accuracies)
```



4 Remarks

As the no. of iterations increases, the log loss decreases which means that the model is improving. The lowest value reached around 0.1 which indicates that the predicted probabilities more accruately at the end of iterations. The accuracy starts at 0.7 and increase gradually reaching to 0.95. It remains relatively stable so its consistent making it more accurate predictions.

[117]:
```
pred1 = (y_pred>=.5)
pred0 = (y_pred<.5)

fig, ax = plt.subplots(figsize=(8, 8))
# true predictions
ax.plot(x_mat[pred1 & (y==1),0],x_mat[pred1 & (y==1),1], 'ro', label='true␣
 ↪positives')
ax.plot(x_mat[pred0 & (y==0),0],x_mat[pred0 & (y==0),1], 'bx', label='true␣
 ↪negatives')
# false predictions
ax.plot(x_mat[pred1 & (y==0),0],x_mat[pred1 & (y==0),1], 'yx', label='false␣
 ↪positives', markersize=15)
ax.plot(x_mat[pred0 & (y==1),0],x_mat[pred0 & (y==1),1], 'yo', label='false␣
 ↪negatives', markersize=15, alpha=.6)
ax.set(title='Truth vs Prediction')
ax.legend(bbox_to_anchor=(1, 0.8), fancybox=True, shadow=True,␣
 ↪fontsize='x-large');
```



5 Remarks

TP - correctly predicted positive values - 25% TN - correctly predicted negative - 70% FP - incorrectly predicted positive values - 0% FN - incorrectly predicted negative values - 5%

Assuming that most of the instances here is from True Negative which means there's a lot of correctly predicted on negative values, the pattern mostly distinguish the negative values of the patterns and 50% of the probability from the pattern is learn to distinguish the positive values.