# Hands_on_Activity_6_1_Modified

| | |
|---|---|
| Technological Institute of the Philippines | Quezon City - Computer Engineering |
| Course Code: | CPE 019 |
| Code Title: | Emerging Technologies in CpE 2 |
| Summer | AY 2024 - 2025 |

| | |
|---|---|
| **Hands-on Activity 6.1** | **Neural Networks** |
| **Name** | Calvadores, Kelly Joseph |
| **Section** | CPE32S1 |
| **Date Performed**: | June 25, 2024 |
| **Date Submitted**: | June 26, 2024 |
| **Date Modified:** | June 28, 2024 |
| **Instructor**: | Engr. Roman M. Richard |

## Activity 6.1 : Neural Networks¶

*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

In [ ]:

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

Define and plot an activation function

### Sigmoid function:¶

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

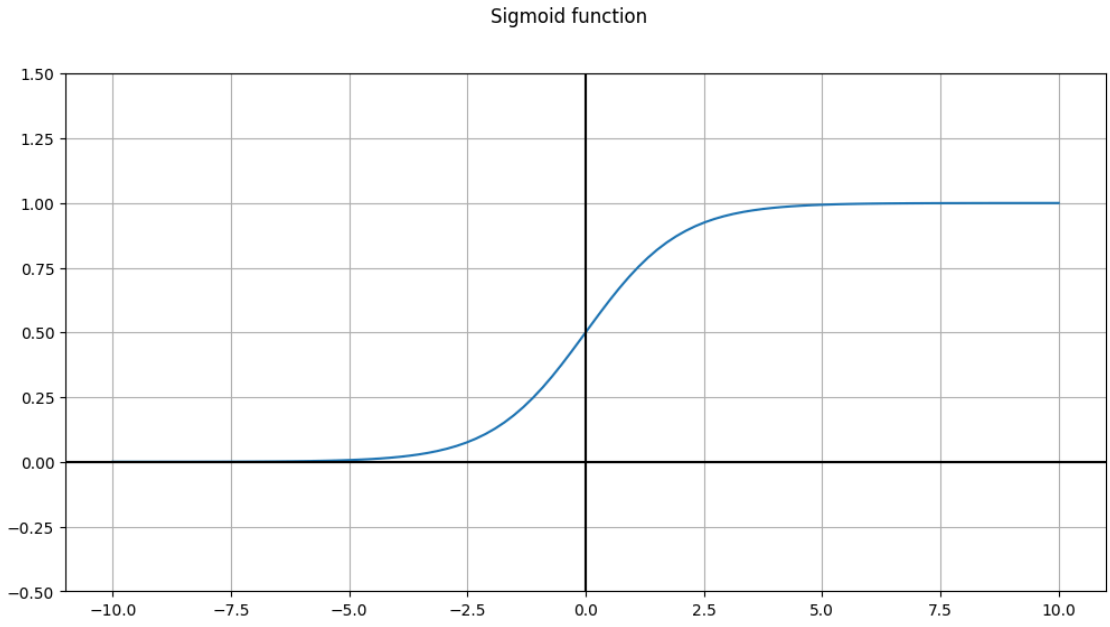
$\sigma$ ranges from (0, 1). When the input $x$ is negative, $\sigma$ is close to 0. When $x$ is positive, $\sigma$ is close to 1. At $x=0$, $\sigma=0.5$

In [ ]:

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

In [ ]:

```
# 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.

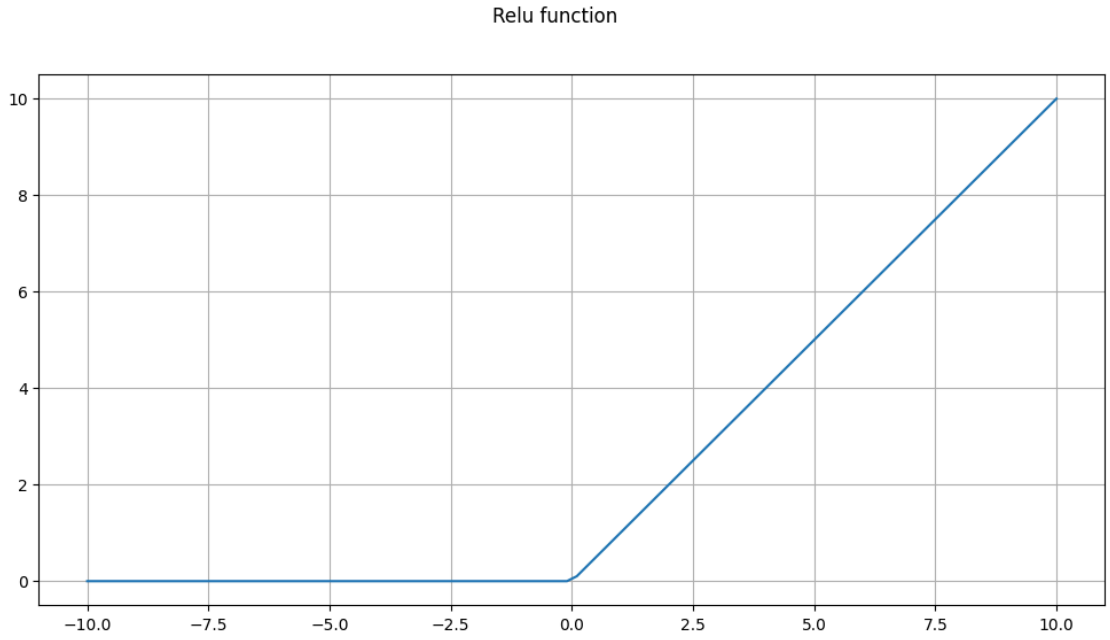
In [ ]:

```
#type your code here
def Relu(x):
    return np.maximum(0,x)
```

Plot the activation function

In [ ]:

```
#type your code here
activationRelu = Relu(vals)
fig = plt.figure(figsize=(12,6))
fig.suptitle('Relu function')
plt.plot(vals, activationRelu)
plt.grid(True, which='both')
```




## Neurons as boolean logic gates¶

### 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 $\left\{0, 1\right\}$, 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 $b=-10$!

In [ ]:

```
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))))
```

In [ ]:

```
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.

### 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.

In [ ]:

```
# Fill in the w1, w2, and b parameters such that the truth table matches
w1 = 10
w2 = 10
b = -15
and_gate = logic_gate(w1, w2, b)

test(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.

In [ ]:

```
#Nor Gate
w1 = -20
w2 = -20
b = 10
nor_gate = logic_gate(w1, w2, b)

test(nor_gate)
```

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

In [ ]:

```
#Nand Gate
w1 = -10
w2 = -10
b = 15
nand_gate = logic_gate(w1, w2, b)

test(nand_gate)
```

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

## Limitation of single neuron¶

Here's the truth table for XOR:

### 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?

In [ ]:

```
# 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
```

## 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.

## 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

In [ ]:

```
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])
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]]
```

## 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

In [ ]:

```
#type your code here
z2 = np.dot(x_in,W_1)
a2 = sigmoid(z2)
z3 = np.dot(a2,z2)
a3 = sigmoid(z3)
z4 = np.dot(a3,z3)

print("The product of array x_in and W_1 (z2) is ", z2)
print("Apply sigmoid function to z2 that results to a2", a2)
print("Get the product of a2 and z2 (z3)", z3)
print("Apply sigmoid function to z3 that results to a3", a3)
print("Get the product of a3 and z3 that results to z4", z4)
```

```
The product of array x_in and W_1 (z2) is  [ 0.8  0.7 -2.1  3.8]
Apply sigmoid function to z2 that results to a2 [0.68997448 0.66818777
0.10909682 0.97811873]
Get the product of a2 and z2 (z3) 4.507458871351723
Apply sigmoid function to z3 that results to a3 0.9890938122523221
Get the product of a3 and z3 that results to z4 4.458299678635824
```

In [ ]:

```
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))
```

1. Apply soft_max_vec function to z4 that results to y_out

In [ ]:

```
#type your code here
y_out = soft_max_vec(z4)
print("The result of applying the soft max vec to z4 is", y_out)
```

```
The result of applying the soft max vec to z4 is 1.0
```

In [ ]:

```
## 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))
```

In [ ]:

```
nn_comp_vec(x_in)
```

Out[ ]:

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

In [ ]:

```
nn_comp_mat(x_mat_in)
```

Out[ ]:

```
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]])
```

## 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

In [ ]:

```
#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?

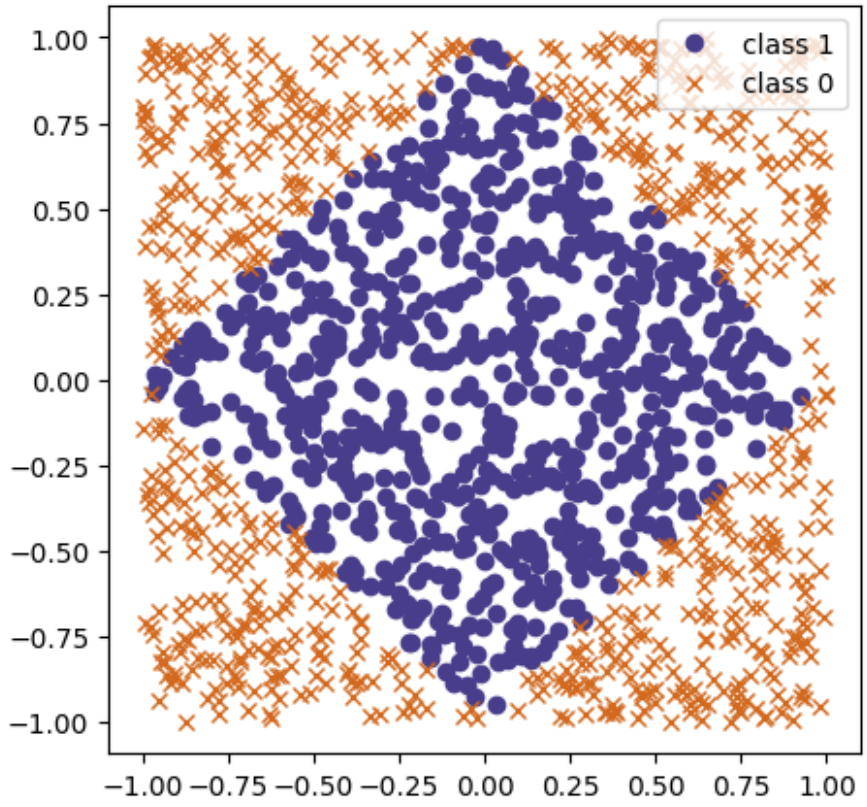
- What learning rates and numbers of iterations worked well?

In [ ]:

```
## 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 = 1500
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 (1500, 3)
shape of y is (1500,)
```

```
<ipython-input-76-36cdab2b66cb>: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-76-36cdab2b66cb>: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')
```




In [ ]:

```
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
```

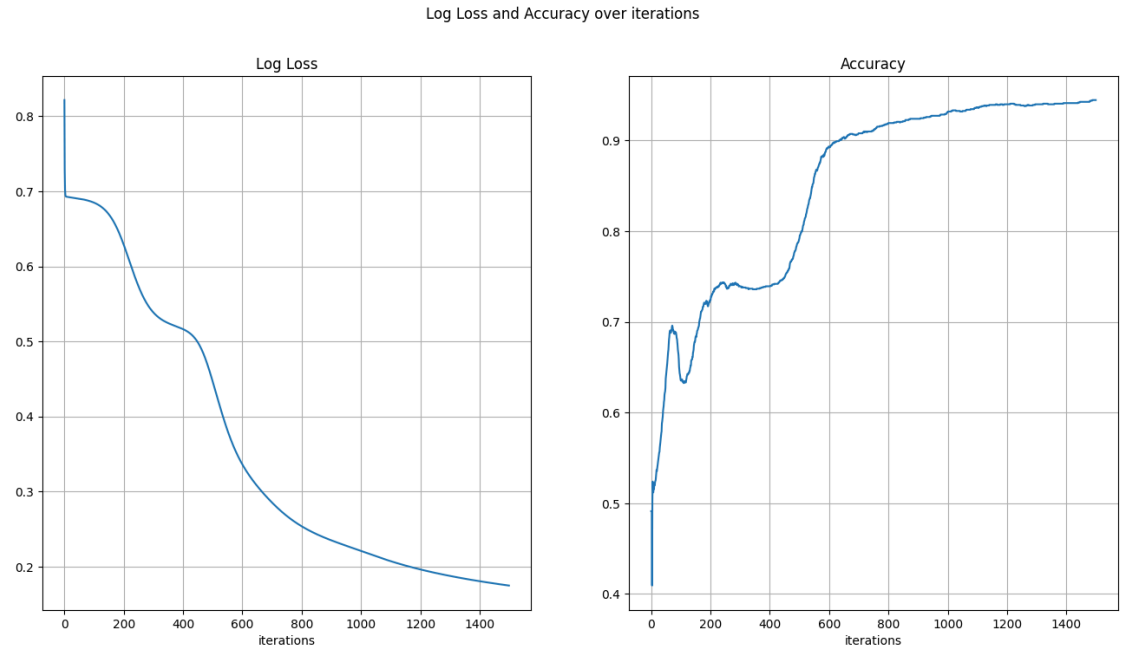
```
    """
    y_pred = np.maximum(y_pred,eps)
    y_pred = np.minimum(y_pred,(1-eps))
    return -(np.sum(y_true * np.log(y_pred)) +
np.sum((1-y_true)*np.log(1-y_pred)))/len(y_true)


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

In [ ]:

```
#### Initialize the network parameters

np.random.seed(1241)

W_1 = np.random.uniform(-1,1,size = (3,4))
W_2 = np.random.uniform(-1,1,size = (4))
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, (GradW_1, GradW_2) = forward_pass(W_1, W_2)

    ## Update the weight matrices
    W_1 = W_1 - learning_rate * GradW_1
    W_2 = W_2 - learning_rate * GradW_2

    ### Compute the loss and accuracy
    Loss = loss_fn(y, y_pred)
    Accuracy = np.sum(y==np.round(y_pred)) / len(y)

    loss_vals.append(Loss)
    accuracies.append(Accuracy)

    ## Print the loss and accuracy for every 200th iteration
    if i % 200 == 0:
        print('iter: {}, loss: {}, accuracy: {}'.format(i, Loss, Accuracy))

plot_loss_accuracy(loss_vals, accuracies)
```

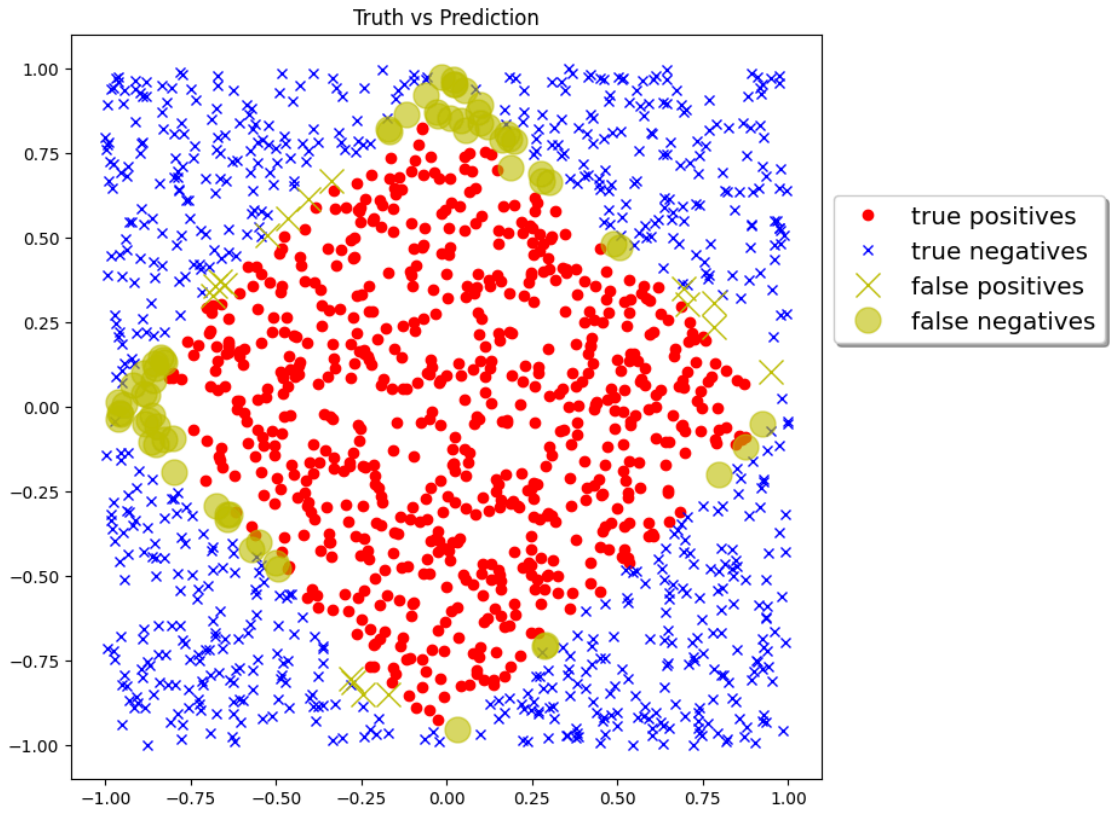
```
iter: 0, loss: 0.8216711004168622, accuracy: 0.49133333333333334
iter: 200, loss: 0.628833879040869, accuracy: 0.724
iter: 400, loss: 0.5163494772382764, accuracy: 0.7393333333333333
iter: 600, loss: 0.3360617158028558, accuracy: 0.8933333333333333
iter: 800, loss: 0.25359911987623773, accuracy: 0.9193333333333333
iter: 1000, loss: 0.2206997299928258, accuracy: 0.9313333333333333
iter: 1200, loss: 0.19609383341122186, accuracy: 0.94
iter: 1400, loss: 0.18039404260932693, accuracy: 0.9413333333333334
```




Plot the predicted answers, with mistakes in yellow

In [ ]:

```
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');
```




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

Which patterns was the neural network able to learn quickly and which took longer?

- The pattern on neural network was able to learn quick is circle pattern, next is center square and Thick right angle took longer than the rest of the pattern. The reason why circle is the quickest to learn is because there is less complexity when learning the circle compare to the Thick Right angle. The Thick Right Angle took longer is because of its sharp edge and an odd shape that it was doing, the thickness add some complexity when learning.

---

What learning rates and numbers of iterations worked well?

- The learning rates that worked well is 0.001, while the number of iteration is 1500.

## 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

In [1]:

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

```
def Tanh(x):
  return np.tanh(x)
```

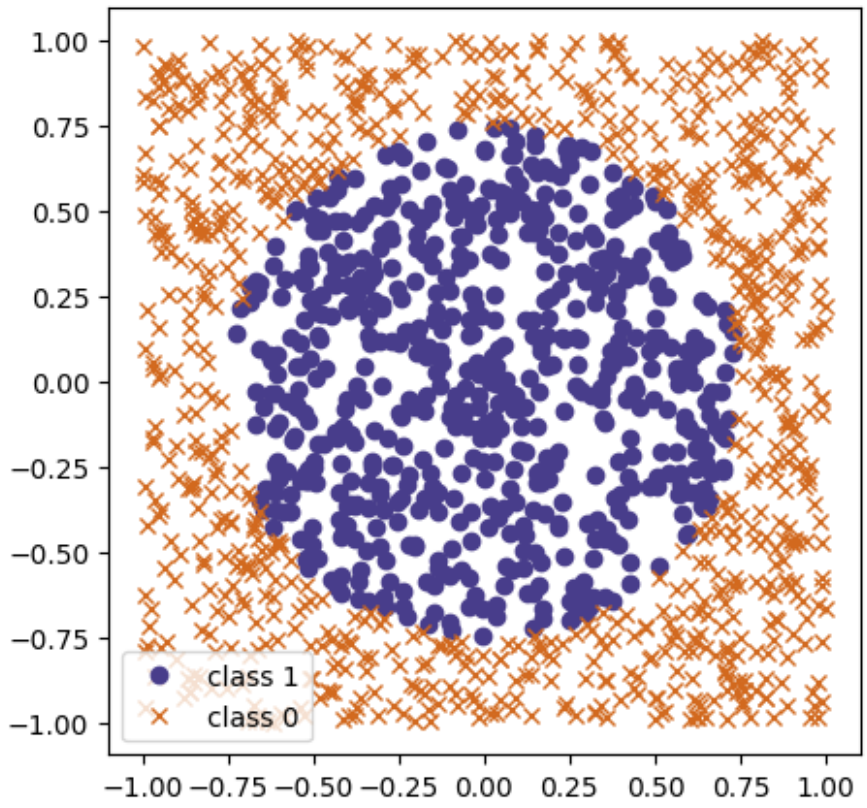
In [12]:

```
num_iter = 1500
learning_rate = 0.001
num_obs = 1500
W_1 = np.random.uniform(-1,1,size = (3,4))
W_2 = np.random.uniform(-1,1,size = (4))
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)
X_mat = X_mat_full
```

In [13]:

```
y = (np.sqrt(X_mat_full[:,0]**2 + X_mat_full[:,1]**2)<.75).astype(int)

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');
```

```
<ipython-input-13-d44ba21d2ed0>:4: 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-13-d44ba21d2ed0>:5: 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')
```




In [14]:

```
def LossFunc(y_true, y_pred, eps=1e-16):
    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)
```

In [22]:

```
def forward_pass(W_1, W_2):
    global X_mat
    global y
    global num_

    z_2 = np.dot(X_mat, W_1)
    a_2 = Tanh(z_2)
    z_3 = np.dot(a_2, W_2)
    y_pred = Tanh(z_3).reshape((len(X_mat),))

    J_z_3_grad = -y + y_pred
    J_W_2_grad = np.dot(J_z_3_grad, a_2)
    a_2_z_2_grad = Tanh(z_2)*(1-Tanh(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 y_pred, gradient
```

In [23]:

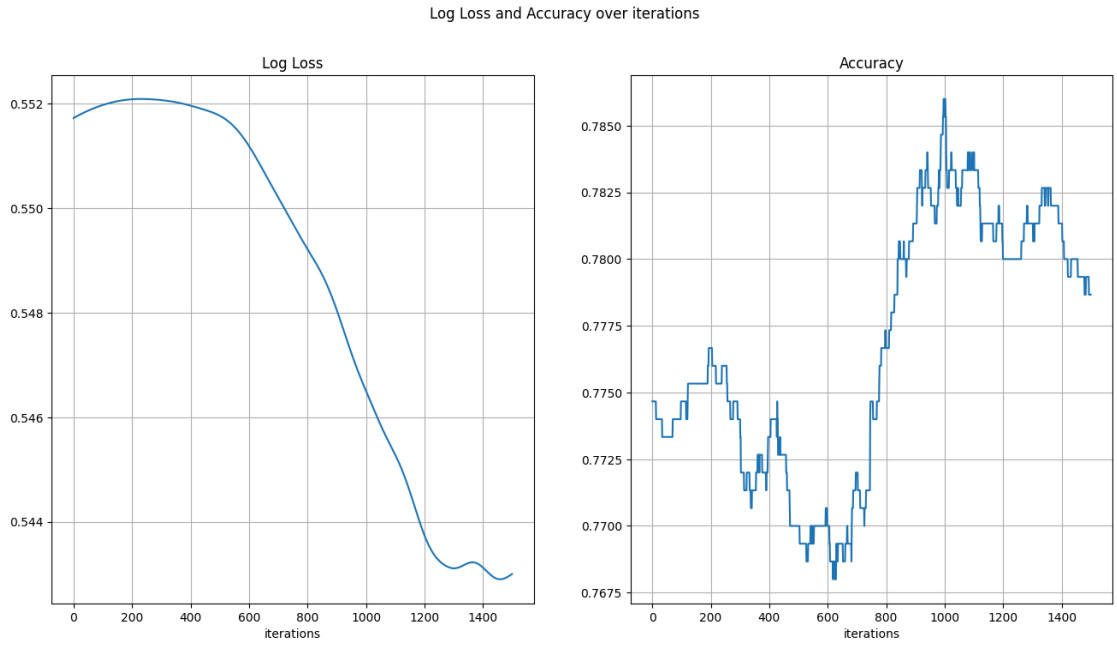
```
def plot_loss_accuracy(loss_vals, accuracies):
    fig = plt.figure(figsize=(16, 8))
    fig.suptitle('Log Loss and Accuracy over iterations')

    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');
```

In [25]:

```
LossVals, Accuracies = [], []
for i in range(num_iter):
    ### Do a forward computation, and get the gradient
    y_pred, (w_1Grad, w_2Grad) = forward_pass(W_1, W_2)

    y_pred = y_pred[:len(y)]

    ## Update the weight matrices
    W_1 = W_1 - learning_rate * w_1Grad
    W_2 = W_2 - learning_rate * w_2Grad

    ### Compute the loss and accuracy
    Loss = LossFunc(y, y_pred)
    LossVals.append(Loss)

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

    ## Print the loss and accuracy for every 300th iteration
    if i % 300 == 0:
      print(f"Iteration {i}: Loss {Loss:.4f}, Accuracy {Accuracy:.4f}")

plot_loss_accuracy(LossVals, Accuracies)
```

```
Iteration 0: Loss 0.5517, Accuracy 0.7747
Iteration 300: Loss 0.5521, Accuracy 0.7740
Iteration 600: Loss 0.5512, Accuracy 0.7700
```

```
Iteration 900: Loss 0.5480, Accuracy 0.7813
Iteration 1200: Loss 0.5437, Accuracy 0.7800
```




## Conclusion¶

- In this activity, I was able to implement the neural network. The idea of neural network is to compute and classify data points into different kind of patterns such as circle, square, diamond, and many more. I was able to apply sigmoid, Relu, and Tanh function and execute it with feedforward and backpropagation as observed the loss and accuracy while training.

In [ ]:

```
!jupyter nbconvert --to html /content/Hands_on_Activity_6_1.ipynb
```

```
[NbConvertApp] Converting notebook /content/Hands_on_Activity_6_1.ipynb to
html
[NbConvertApp] Writing 1272594 bytes to /content/Hands_on_Activity_6_1.html
```

In [ ]:

```
!pandoc /content/Hands_on_Activity_6_1.html -s -o
/content/Hands_on_Activity_6_1.docx
```

```
[WARNING] Duplicate identifier 'Exercise' at input line 15515 column 77
[WARNING] Duplicate identifier 'Backpropagation' at input line 16342 column
98
```