Hands_on_Activity_6_1

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
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 [26]:

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

Define and plot an activation function

Sigmoid function:¶

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

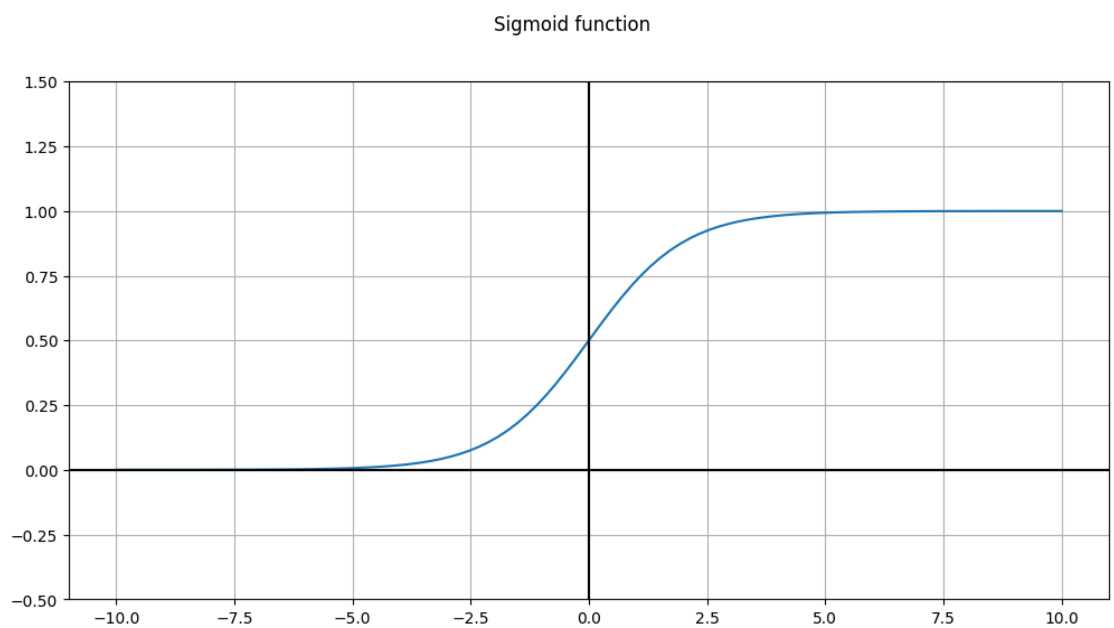
σ ranges from (0, 1). When the input x is negative, σ is close to 0. When x is positive, σ is close to 1. At $x=0$, $\sigma=0.5$

In [27]:

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

In [28]:

```
# 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.

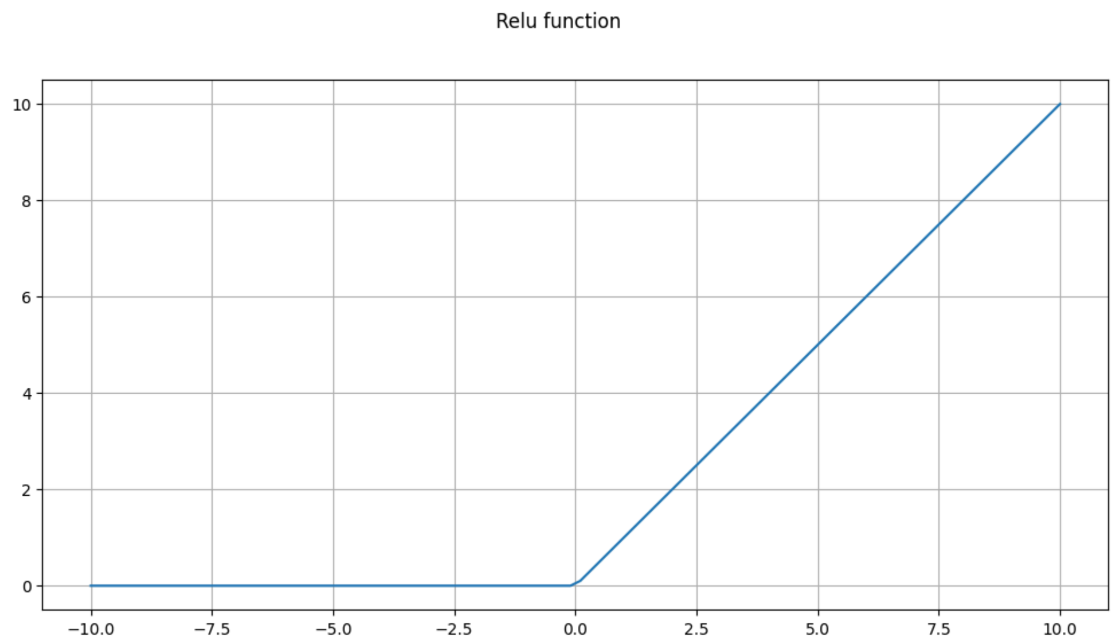
In [29]:

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

Plot the activation function

In [30]:

```
#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 [31]:

```
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 [32]:

```
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 [33]:

```
# 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 [34]:

```
#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 [35]:

```
#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 [36]:

```
# 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 [37]:

```
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 [38]:

```
#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 [39]:

```
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 [40]:

```
#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 [41]:

```
## 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 [42]:

```
nn_comp_vec(x_in)
```

Out[42]:

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

In [43]:

```
nn_comp_mat(x_mat_in)
```

Out[43]:

```
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 [44]:

```
#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?

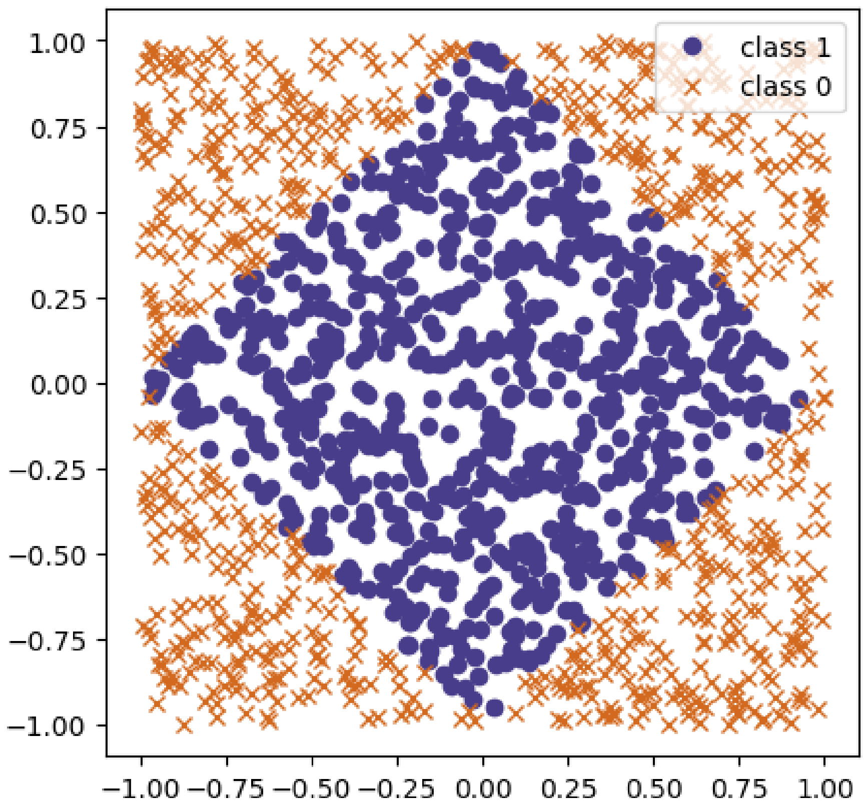
- What learning rates and numbers of iterations worked well?

In [76]:

```
## 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 [77]:

```
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
```

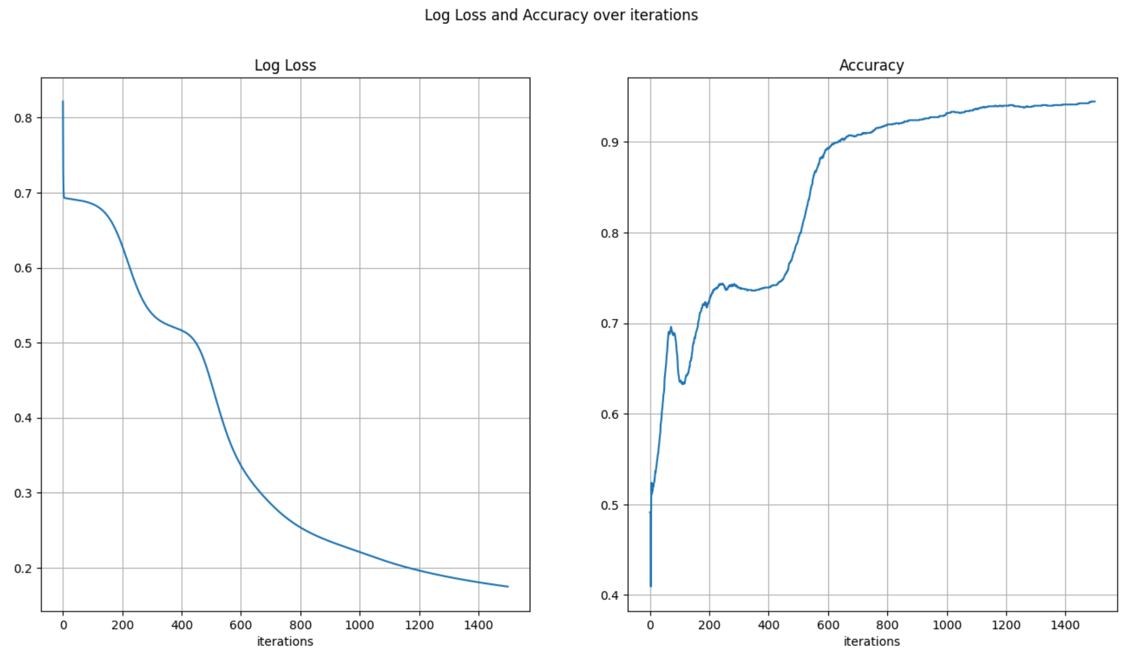
```
    """
    y_pred = np.maximum(y_pred,eps)
    y_pred = np.minimum(y_pred,(1-eps))
    return -(np.sum(y_true * np.log(y_pred)) +
np.sum((1-y_true)*np.log(1-y_pred)))/len(y_true)


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 [78]:

```
#### 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)
```

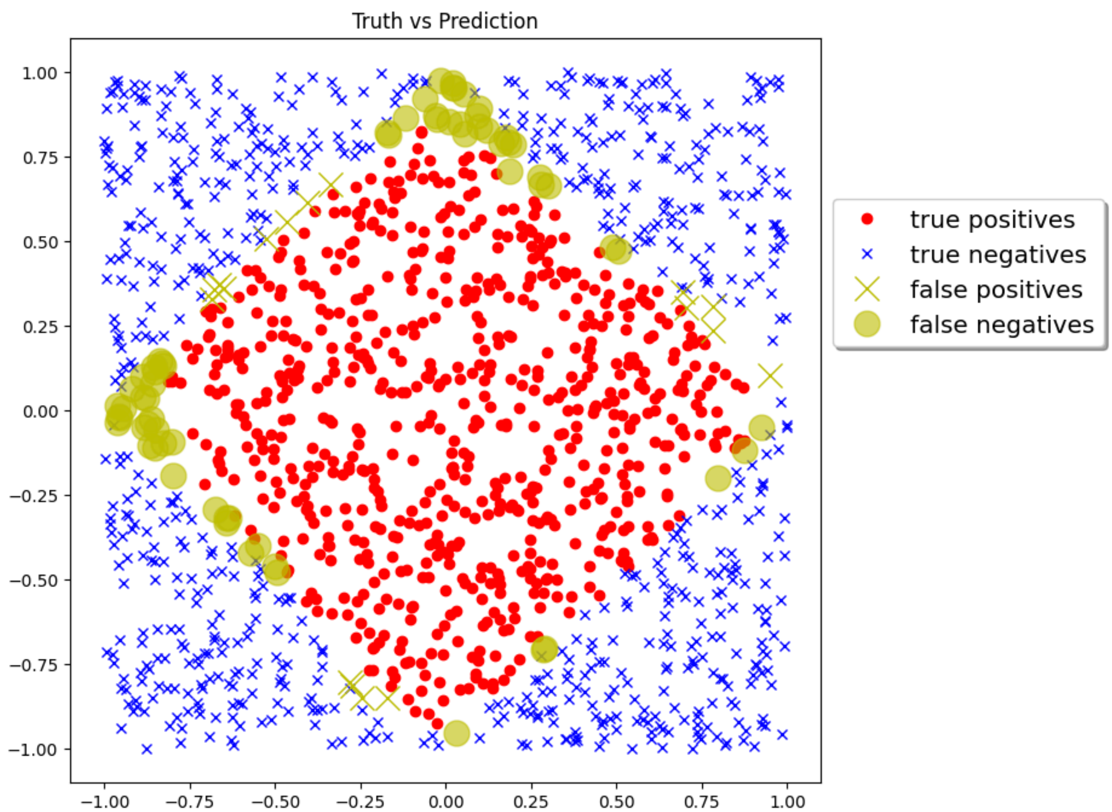
```
iter: 0, loss: 0.8216711004168622, accuracy: 0.49133333333333334
iter: 200, loss: 0.628833879040869, accuracy: 0.724
iter: 400, loss: 0.5163494772382764, accuracy: 0.7393333333333333
iter: 600, loss: 0.3360617158028558, accuracy: 0.8933333333333333
iter: 800, loss: 0.25359911987623773, accuracy: 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 [79]:

```
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

Activation Function¶

In [180]:

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

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

In [181]:

```
num_iter = 300
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
```

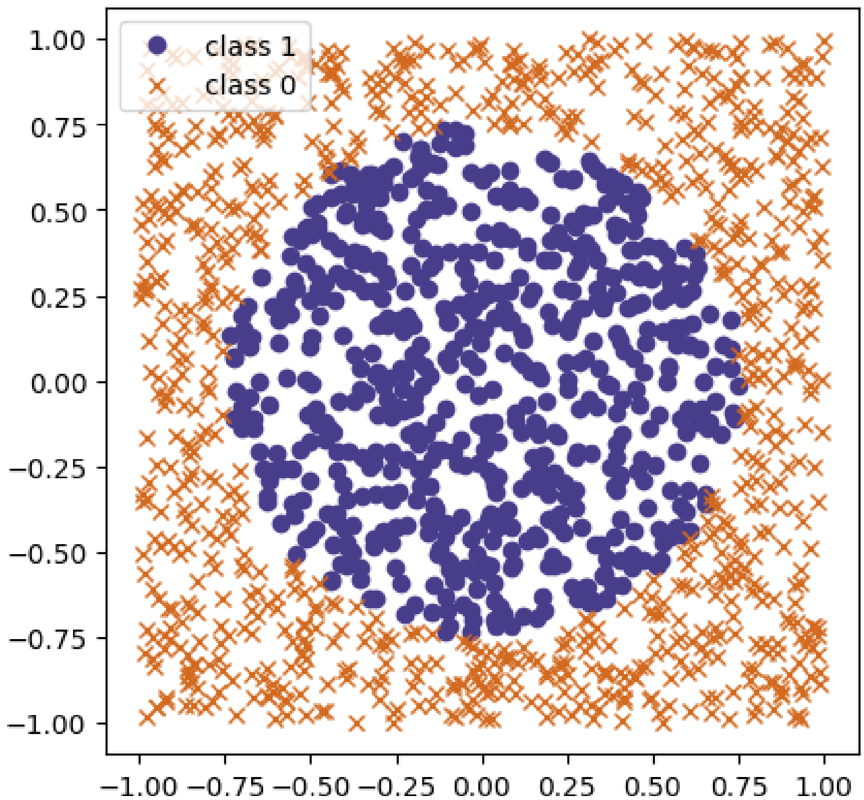
Feedforward¶

Backpropagation¶

In [182]:

```
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-182-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-182-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 [183]:

```
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 [184]:

```
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-Relu(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 [185]:

```
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 [186]:

```
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 200th iteration
    if i % 200:
      print(f"Iteration {i}: Loss {Loss:.4f}, Accuracy {Accuracy:.4f}")

plot_loss_accuracy(LossVals, Accuracies)
```

```
Iteration 1: Loss 12.0621, Accuracy 0.5727
Iteration 2: Loss 2.3201, Accuracy 0.5047
Iteration 3: Loss 2.2110, Accuracy 0.5147
Iteration 4: Loss 0.7897, Accuracy 0.5147
```

```
Iteration 5: Loss 0.7408, Accuracy 0.5220
Iteration 6: Loss 0.7176, Accuracy 0.5180
Iteration 7: Loss 0.7047, Accuracy 0.5140
Iteration 8: Loss 0.6972, Accuracy 0.5373
Iteration 9: Loss 0.6927, Accuracy 0.5480
Iteration 10: Loss 0.6901, Accuracy 0.5607
Iteration 11: Loss 0.6886, Accuracy 0.5620
Iteration 12: Loss 0.6877, Accuracy 0.5613
Iteration 13: Loss 0.6871, Accuracy 0.5580
Iteration 14: Loss 0.6867, Accuracy 0.5613
Iteration 15: Loss 0.6864, Accuracy 0.5647
Iteration 16: Loss 0.6862, Accuracy 0.5647
Iteration 17: Loss 0.6861, Accuracy 0.5627
Iteration 18: Loss 0.6859, Accuracy 0.5647
Iteration 19: Loss 0.6858, Accuracy 0.5660
Iteration 20: Loss 0.6856, Accuracy 0.5640
Iteration 21: Loss 0.6855, Accuracy 0.5647
Iteration 22: Loss 0.6854, Accuracy 0.5647
Iteration 23: Loss 0.6853, Accuracy 0.5640
Iteration 24: Loss 0.6852, Accuracy 0.5680
Iteration 25: Loss 0.6851, Accuracy 0.5687
Iteration 26: Loss 0.6850, Accuracy 0.5693
Iteration 27: Loss 0.6850, Accuracy 0.5693
Iteration 28: Loss 0.6849, Accuracy 0.5700
Iteration 29: Loss 0.6848, Accuracy 0.5700
Iteration 30: Loss 0.6847, Accuracy 0.5707
Iteration 31: Loss 0.6847, Accuracy 0.5700
Iteration 32: Loss 0.6846, Accuracy 0.5713
Iteration 33: Loss 0.6845, Accuracy 0.5713
Iteration 34: Loss 0.6845, Accuracy 0.5707
Iteration 35: Loss 0.6844, Accuracy 0.5713
Iteration 36: Loss 0.6844, Accuracy 0.5700
Iteration 37: Loss 0.6843, Accuracy 0.5707
Iteration 38: Loss 0.6843, Accuracy 0.5720
Iteration 39: Loss 0.6842, Accuracy 0.5720
Iteration 40: Loss 0.6842, Accuracy 0.5727
Iteration 41: Loss 0.6841, Accuracy 0.5727
Iteration 42: Loss 0.6841, Accuracy 0.5720
Iteration 43: Loss 0.6841, Accuracy 0.5720
Iteration 44: Loss 0.6840, Accuracy 0.5733
Iteration 45: Loss 0.6840, Accuracy 0.5727
Iteration 46: Loss 0.6839, Accuracy 0.5727
Iteration 47: Loss 0.6839, Accuracy 0.5727
Iteration 48: Loss 0.6839, Accuracy 0.5720
Iteration 49: Loss 0.6839, Accuracy 0.5720
Iteration 50: Loss 0.6838, Accuracy 0.5727
Iteration 51: Loss 0.6838, Accuracy 0.5727
Iteration 52: Loss 0.6838, Accuracy 0.5720
Iteration 53: Loss 0.6837, Accuracy 0.5727
Iteration 54: Loss 0.6837, Accuracy 0.5727
```

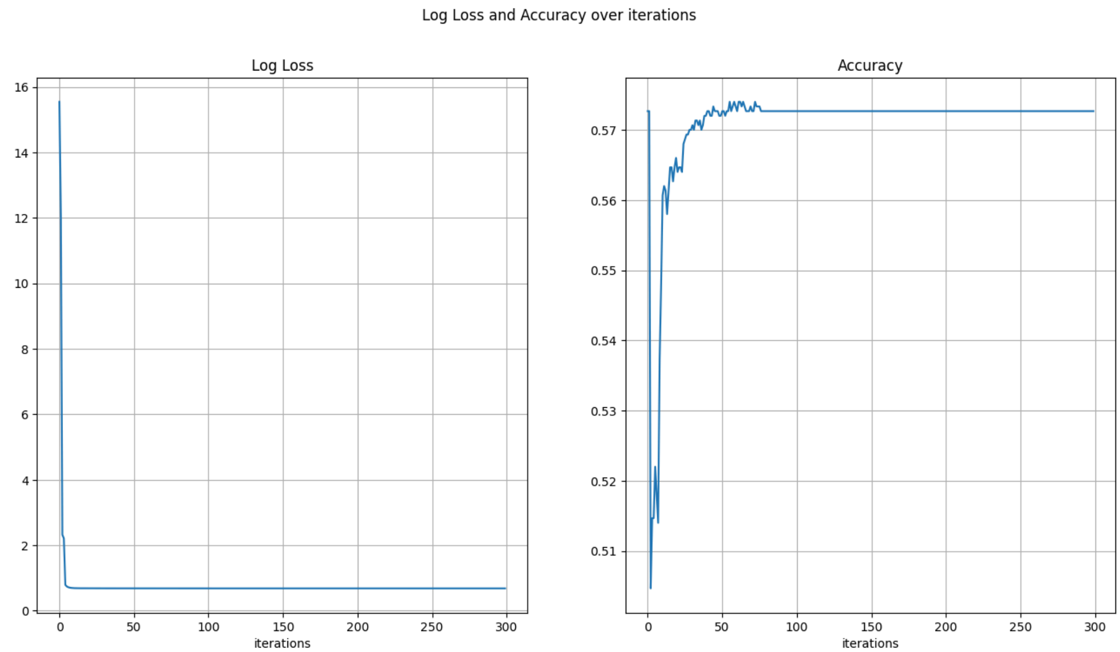
```
Iteration 55: Loss 0.6837, Accuracy 0.5740
Iteration 56: Loss 0.6837, Accuracy 0.5727
Iteration 57: Loss 0.6837, Accuracy 0.5733
Iteration 58: Loss 0.6836, Accuracy 0.5740
Iteration 59: Loss 0.6836, Accuracy 0.5733
Iteration 60: Loss 0.6836, Accuracy 0.5727
Iteration 61: Loss 0.6836, Accuracy 0.5740
Iteration 62: Loss 0.6836, Accuracy 0.5740
Iteration 63: Loss 0.6835, Accuracy 0.5733
Iteration 64: Loss 0.6835, Accuracy 0.5740
Iteration 65: Loss 0.6835, Accuracy 0.5733
Iteration 66: Loss 0.6835, Accuracy 0.5727
Iteration 67: Loss 0.6835, Accuracy 0.5727
Iteration 68: Loss 0.6835, Accuracy 0.5727
Iteration 69: Loss 0.6835, Accuracy 0.5733
Iteration 70: Loss 0.6834, Accuracy 0.5727
Iteration 71: Loss 0.6834, Accuracy 0.5727
Iteration 72: Loss 0.6834, Accuracy 0.5740
Iteration 73: Loss 0.6834, Accuracy 0.5733
Iteration 74: Loss 0.6834, Accuracy 0.5733
Iteration 75: Loss 0.6834, Accuracy 0.5733
Iteration 76: Loss 0.6834, Accuracy 0.5727
Iteration 77: Loss 0.6834, Accuracy 0.5727
Iteration 78: Loss 0.6834, Accuracy 0.5727
Iteration 79: Loss 0.6833, Accuracy 0.5727
Iteration 80: Loss 0.6833, Accuracy 0.5727
Iteration 81: Loss 0.6833, Accuracy 0.5727
Iteration 82: Loss 0.6833, Accuracy 0.5727
Iteration 83: Loss 0.6833, Accuracy 0.5727
Iteration 84: Loss 0.6833, Accuracy 0.5727
Iteration 85: Loss 0.6833, Accuracy 0.5727
Iteration 86: Loss 0.6833, Accuracy 0.5727
Iteration 87: Loss 0.6833, Accuracy 0.5727
Iteration 88: Loss 0.6833, Accuracy 0.5727
Iteration 89: Loss 0.6833, Accuracy 0.5727
Iteration 90: Loss 0.6833, Accuracy 0.5727
Iteration 91: Loss 0.6833, Accuracy 0.5727
Iteration 92: Loss 0.6833, Accuracy 0.5727
Iteration 93: Loss 0.6832, Accuracy 0.5727
Iteration 94: Loss 0.6832, Accuracy 0.5727
Iteration 95: Loss 0.6832, Accuracy 0.5727
Iteration 96: Loss 0.6832, Accuracy 0.5727
Iteration 97: Loss 0.6832, Accuracy 0.5727
Iteration 98: Loss 0.6832, Accuracy 0.5727
Iteration 99: Loss 0.6832, Accuracy 0.5727
Iteration 100: Loss 0.6832, Accuracy 0.5727
Iteration 101: Loss 0.6832, Accuracy 0.5727
Iteration 102: Loss 0.6832, Accuracy 0.5727
Iteration 103: Loss 0.6832, Accuracy 0.5727
Iteration 104: Loss 0.6832, Accuracy 0.5727
```

```
Iteration 105: Loss 0.6832, Accuracy 0.5727
Iteration 106: Loss 0.6832, Accuracy 0.5727
Iteration 107: Loss 0.6832, Accuracy 0.5727
Iteration 108: Loss 0.6832, Accuracy 0.5727
Iteration 109: Loss 0.6832, Accuracy 0.5727
Iteration 110: Loss 0.6832, Accuracy 0.5727
Iteration 111: Loss 0.6832, Accuracy 0.5727
Iteration 112: Loss 0.6832, Accuracy 0.5727
Iteration 113: Loss 0.6832, Accuracy 0.5727
Iteration 114: Loss 0.6832, Accuracy 0.5727
Iteration 115: Loss 0.6832, Accuracy 0.5727
Iteration 116: Loss 0.6832, Accuracy 0.5727
Iteration 117: Loss 0.6832, Accuracy 0.5727
Iteration 118: Loss 0.6832, Accuracy 0.5727
Iteration 119: Loss 0.6832, Accuracy 0.5727
Iteration 120: Loss 0.6832, Accuracy 0.5727
Iteration 121: Loss 0.6832, Accuracy 0.5727
Iteration 122: Loss 0.6831, Accuracy 0.5727
Iteration 123: Loss 0.6831, Accuracy 0.5727
Iteration 124: Loss 0.6831, Accuracy 0.5727
Iteration 125: Loss 0.6831, Accuracy 0.5727
Iteration 126: Loss 0.6831, Accuracy 0.5727
Iteration 127: Loss 0.6831, Accuracy 0.5727
Iteration 128: Loss 0.6831, Accuracy 0.5727
Iteration 129: Loss 0.6831, Accuracy 0.5727
Iteration 130: Loss 0.6831, Accuracy 0.5727
Iteration 131: Loss 0.6831, Accuracy 0.5727
Iteration 132: Loss 0.6831, Accuracy 0.5727
Iteration 133: Loss 0.6831, Accuracy 0.5727
Iteration 134: Loss 0.6831, Accuracy 0.5727
Iteration 135: Loss 0.6831, Accuracy 0.5727
Iteration 136: Loss 0.6831, Accuracy 0.5727
Iteration 137: Loss 0.6831, Accuracy 0.5727
Iteration 138: Loss 0.6831, Accuracy 0.5727
Iteration 139: Loss 0.6831, Accuracy 0.5727
Iteration 140: Loss 0.6831, Accuracy 0.5727
Iteration 141: Loss 0.6831, Accuracy 0.5727
Iteration 142: Loss 0.6831, Accuracy 0.5727
Iteration 143: Loss 0.6831, Accuracy 0.5727
Iteration 144: Loss 0.6831, Accuracy 0.5727
Iteration 145: Loss 0.6831, Accuracy 0.5727
Iteration 146: Loss 0.6831, Accuracy 0.5727
Iteration 147: Loss 0.6831, Accuracy 0.5727
Iteration 148: Loss 0.6831, Accuracy 0.5727
Iteration 149: Loss 0.6831, Accuracy 0.5727
Iteration 150: Loss 0.6831, Accuracy 0.5727
Iteration 151: Loss 0.6831, Accuracy 0.5727
Iteration 152: Loss 0.6831, Accuracy 0.5727
Iteration 153: Loss 0.6831, Accuracy 0.5727
Iteration 154: Loss 0.6831, Accuracy 0.5727
```

```
Iteration 155: Loss 0.6831, Accuracy 0.5727
Iteration 156: Loss 0.6831, Accuracy 0.5727
Iteration 157: Loss 0.6831, Accuracy 0.5727
Iteration 158: Loss 0.6831, Accuracy 0.5727
Iteration 159: Loss 0.6831, Accuracy 0.5727
Iteration 160: Loss 0.6831, Accuracy 0.5727
Iteration 161: Loss 0.6831, Accuracy 0.5727
Iteration 162: Loss 0.6831, Accuracy 0.5727
Iteration 163: Loss 0.6831, Accuracy 0.5727
Iteration 164: Loss 0.6831, Accuracy 0.5727
Iteration 165: Loss 0.6831, Accuracy 0.5727
Iteration 166: Loss 0.6831, Accuracy 0.5727
Iteration 167: Loss 0.6831, Accuracy 0.5727
Iteration 168: Loss 0.6831, Accuracy 0.5727
Iteration 169: Loss 0.6831, Accuracy 0.5727
Iteration 170: Loss 0.6831, Accuracy 0.5727
Iteration 171: Loss 0.6831, Accuracy 0.5727
Iteration 172: Loss 0.6831, Accuracy 0.5727
Iteration 173: Loss 0.6831, Accuracy 0.5727
Iteration 174: Loss 0.6831, Accuracy 0.5727
Iteration 175: Loss 0.6831, Accuracy 0.5727
Iteration 176: Loss 0.6831, Accuracy 0.5727
Iteration 177: Loss 0.6831, Accuracy 0.5727
Iteration 178: Loss 0.6831, Accuracy 0.5727
Iteration 179: Loss 0.6831, Accuracy 0.5727
Iteration 180: Loss 0.6831, Accuracy 0.5727
Iteration 181: Loss 0.6831, Accuracy 0.5727
Iteration 182: Loss 0.6831, Accuracy 0.5727
Iteration 183: Loss 0.6831, Accuracy 0.5727
Iteration 184: Loss 0.6831, Accuracy 0.5727
Iteration 185: Loss 0.6831, Accuracy 0.5727
Iteration 186: Loss 0.6831, Accuracy 0.5727
Iteration 187: Loss 0.6831, Accuracy 0.5727
Iteration 188: Loss 0.6831, Accuracy 0.5727
Iteration 189: Loss 0.6831, Accuracy 0.5727
Iteration 190: Loss 0.6831, Accuracy 0.5727
Iteration 191: Loss 0.6831, Accuracy 0.5727
Iteration 192: Loss 0.6831, Accuracy 0.5727
Iteration 193: Loss 0.6831, Accuracy 0.5727
Iteration 194: Loss 0.6831, Accuracy 0.5727
Iteration 195: Loss 0.6831, Accuracy 0.5727
Iteration 196: Loss 0.6831, Accuracy 0.5727
Iteration 197: Loss 0.6831, Accuracy 0.5727
Iteration 198: Loss 0.6831, Accuracy 0.5727
Iteration 199: Loss 0.6831, Accuracy 0.5727
Iteration 201: Loss 0.6831, Accuracy 0.5727
Iteration 202: Loss 0.6831, Accuracy 0.5727
Iteration 203: Loss 0.6831, Accuracy 0.5727
Iteration 204: Loss 0.6831, Accuracy 0.5727
Iteration 205: Loss 0.6831, Accuracy 0.5727
```

```
Iteration 206: Loss 0.6831, Accuracy 0.5727
Iteration 207: Loss 0.6831, Accuracy 0.5727
Iteration 208: Loss 0.6831, Accuracy 0.5727
Iteration 209: Loss 0.6831, Accuracy 0.5727
Iteration 210: Loss 0.6831, Accuracy 0.5727
Iteration 211: Loss 0.6831, Accuracy 0.5727
Iteration 212: Loss 0.6831, Accuracy 0.5727
Iteration 213: Loss 0.6831, Accuracy 0.5727
Iteration 214: Loss 0.6831, Accuracy 0.5727
Iteration 215: Loss 0.6831, Accuracy 0.5727
Iteration 216: Loss 0.6831, Accuracy 0.5727
Iteration 217: Loss 0.6831, Accuracy 0.5727
Iteration 218: Loss 0.6831, Accuracy 0.5727
Iteration 219: Loss 0.6831, Accuracy 0.5727
Iteration 220: Loss 0.6831, Accuracy 0.5727
Iteration 221: Loss 0.6831, Accuracy 0.5727
Iteration 222: Loss 0.6831, Accuracy 0.5727
Iteration 223: Loss 0.6831, Accuracy 0.5727
Iteration 224: Loss 0.6831, Accuracy 0.5727
Iteration 225: Loss 0.6831, Accuracy 0.5727
Iteration 226: Loss 0.6831, Accuracy 0.5727
Iteration 227: Loss 0.6831, Accuracy 0.5727
Iteration 228: Loss 0.6831, Accuracy 0.5727
Iteration 229: Loss 0.6831, Accuracy 0.5727
Iteration 230: Loss 0.6831, Accuracy 0.5727
Iteration 231: Loss 0.6831, Accuracy 0.5727
Iteration 232: Loss 0.6830, Accuracy 0.5727
Iteration 233: Loss 0.6830, Accuracy 0.5727
Iteration 234: Loss 0.6830, Accuracy 0.5727
Iteration 235: Loss 0.6830, Accuracy 0.5727
Iteration 236: Loss 0.6830, Accuracy 0.5727
Iteration 237: Loss 0.6830, Accuracy 0.5727
Iteration 238: Loss 0.6830, Accuracy 0.5727
Iteration 239: Loss 0.6830, Accuracy 0.5727
Iteration 240: Loss 0.6830, Accuracy 0.5727
Iteration 241: Loss 0.6830, Accuracy 0.5727
Iteration 242: Loss 0.6830, Accuracy 0.5727
Iteration 243: Loss 0.6830, Accuracy 0.5727
Iteration 244: Loss 0.6830, Accuracy 0.5727
Iteration 245: Loss 0.6830, Accuracy 0.5727
Iteration 246: Loss 0.6830, Accuracy 0.5727
Iteration 247: Loss 0.6830, Accuracy 0.5727
Iteration 248: Loss 0.6830, Accuracy 0.5727
Iteration 249: Loss 0.6830, Accuracy 0.5727
Iteration 250: Loss 0.6830, Accuracy 0.5727
Iteration 251: Loss 0.6830, Accuracy 0.5727
Iteration 252: Loss 0.6830, Accuracy 0.5727
Iteration 253: Loss 0.6830, Accuracy 0.5727
Iteration 254: Loss 0.6830, Accuracy 0.5727
Iteration 255: Loss 0.6830, Accuracy 0.5727
```

```
Iteration 256: Loss 0.6830, Accuracy 0.5727
Iteration 257: Loss 0.6830, Accuracy 0.5727
Iteration 258: Loss 0.6830, Accuracy 0.5727
Iteration 259: Loss 0.6830, Accuracy 0.5727
Iteration 260: Loss 0.6830, Accuracy 0.5727
Iteration 261: Loss 0.6830, Accuracy 0.5727
Iteration 262: Loss 0.6830, Accuracy 0.5727
Iteration 263: Loss 0.6830, Accuracy 0.5727
Iteration 264: Loss 0.6830, Accuracy 0.5727
Iteration 265: Loss 0.6830, Accuracy 0.5727
Iteration 266: Loss 0.6830, Accuracy 0.5727
Iteration 267: Loss 0.6830, Accuracy 0.5727
Iteration 268: Loss 0.6830, Accuracy 0.5727
Iteration 269: Loss 0.6830, Accuracy 0.5727
Iteration 270: Loss 0.6830, Accuracy 0.5727
Iteration 271: Loss 0.6830, Accuracy 0.5727
Iteration 272: Loss 0.6830, Accuracy 0.5727
Iteration 273: Loss 0.6830, Accuracy 0.5727
Iteration 274: Loss 0.6830, Accuracy 0.5727
Iteration 275: Loss 0.6830, Accuracy 0.5727
Iteration 276: Loss 0.6830, Accuracy 0.5727
Iteration 277: Loss 0.6830, Accuracy 0.5727
Iteration 278: Loss 0.6830, Accuracy 0.5727
Iteration 279: Loss 0.6830, Accuracy 0.5727
Iteration 280: Loss 0.6830, Accuracy 0.5727
Iteration 281: Loss 0.6830, Accuracy 0.5727
Iteration 282: Loss 0.6830, Accuracy 0.5727
Iteration 283: Loss 0.6830, Accuracy 0.5727
Iteration 284: Loss 0.6830, Accuracy 0.5727
Iteration 285: Loss 0.6830, Accuracy 0.5727
Iteration 286: Loss 0.6830, Accuracy 0.5727
Iteration 287: Loss 0.6830, Accuracy 0.5727
Iteration 288: Loss 0.6830, Accuracy 0.5727
Iteration 289: Loss 0.6830, Accuracy 0.5727
Iteration 290: Loss 0.6830, Accuracy 0.5727
Iteration 291: Loss 0.6830, Accuracy 0.5727
Iteration 292: Loss 0.6830, Accuracy 0.5727
Iteration 293: Loss 0.6830, Accuracy 0.5727
Iteration 294: Loss 0.6830, Accuracy 0.5727
Iteration 295: Loss 0.6830, Accuracy 0.5727
Iteration 296: Loss 0.6830, Accuracy 0.5727
Iteration 297: Loss 0.6830, Accuracy 0.5727
Iteration 298: Loss 0.6830, Accuracy 0.5727
Iteration 299: Loss 0.6830, Accuracy 0.5727
```



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 functions and execute them with feedforward and backpropagation as observed the loss and accuracy while training.

In [187]:

```
!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 1270085 bytes to /content/Hands_on_Activity_6_1.html
```

In [188]:

```
!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
```

In [ ]: