Hands\_on\_Activity\_6\_1\_Neural\_Networks\_Ramos

|  |  |
| --- | --- |
| Technological Institute of the Philippines | Quezon City - Computer Engineering |
| Course Code: | CPE 019 |
| Code Title: | Emerging Technologies in CpE 2 - Fundamentals of Data Science |
| 1st Semester | AY 2023-2024 |
|  |  |
| **ACTIVITY NO.** | **Hands-on Activity 6.1 Neural Networks** |
| **Name** | William Laurence M. Ramos |
| **Section** | CPE32S1 |
| **Date Performed**: | 28/06/2024 |
| **Date Submitted**: | 28/06/2024 |
| **Instructor**: | Engr. Roman M. Richard |

# Hands-on Activity 6.1 Neural Networks[¶](#Hands-on-Activity-6.1-Neural-Networks)

#### Objective(s):[¶](#Objective(s):)

This activity aims to demonstrate the concepts of neural networks

#### Intended Learning Outcomes (ILOs):[¶](#Intended-Learning-Outcomes-(ILOs):)

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

#### Resources:[¶](#Resources:)

* Jupyter Notebook

#### Procedure:[¶](#Procedure:)

Import the libraries

In [ ]:

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

Define and plot an activation function

### Sigmoid 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]);

![](data:image/png;base64;base64,)

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

![image.png](data:image/png;base64;base64,)

In [ ]:

# Defining the tanh Function  
def tanh(x):  
 """Hyperbolic tangent (tanh) function"""  
 return np.tanh(x)

* For this part i have choosen the ReLU activation function

Plot the activation function

In [ ]:

# Generate values for x-axis  
vals = np.linspace(-10, 10, num=100, dtype=np.float32)  
  
# Compute the tanh activation for the generated values  
activation = tanh(vals)  
  
# Plot tanh activation function  
fig = plt.figure(figsize=(12,6))  
fig.suptitle('Tanh function')  
plt.plot(vals, activation)  
plt.grid(True, which='both')  
plt.axhline(y=0, color='k')  
plt.axvline(x=0, color='k')  
plt.ylim([-1.5, 1.5])  
plt.show()

![](data:image/png;base64;base64,)

### Neurons as boolean logic gates[¶](#Neurons-as-boolean-logic-gates)

### OR Gate[¶](#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)

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\_nor = -1  
w2\_nor = -1  
b\_nor = 1.5  
nor\_gate = logic\_gate(w1\_nor, w2\_nor, b\_nor)  
print ("NOR GATE: ")  
test(nor\_gate)

NOR GATE:   
0, 0: 1.0  
0, 1: 1.0  
1, 0: 1.0  
1, 1: 0.0

In [ ]:

# NAND Gate  
w1\_nand = -10  
w2\_nand= -10  
b\_nand = 15  
nand\_gate = logic\_gate(w1\_nand, w2\_nand, b\_nand)  
print ("NAND GATE: ")  
test(nand\_gate)

NAND GATE:   
0, 0: 1.0  
0, 1: 1.0  
1, 0: 1.0  
1, 1: 0.0

## Limitation of single neuron[¶](#Limitation-of-single-neuron)

Here's the truth table for XOR:

### XOR (Exclusive Or) Gate[¶](#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[¶](#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[¶](#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[¶](#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 [ ]:

# 1 Get the product of an array x\_in and W\_1 (z2)  
z2 = np.dot(x\_in, W\_1)  
  
# 2 Apply the sigmoid function go z2 that results to a2  
a2 = sigmoid(z2)  
  
# 3 Get the product of a2 and z2 (z3)  
z3 = np.dot(a2, z2)  
  
# 4 Apply the sigmoud function to z3 that resutls to a3  
a3 = sigmoid(z3)  
  
# 5 get the product of a3 and z3 that results to z4  
z4 = np.dot(a3, z3)  
  
# printing the result (z4)  
print(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)

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[¶](#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? </br> All of the given patterns below executes and learn quiclky. all having the same execusion time.
* What learning rates and numbers of iterations worked well? </br> Based on the proceeding activity the learning rate is ajusted depending on the graph or plotting of the losses. A learning rate of 0.0007 is set and the epoch to 5000

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 = 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-20-e9ff310557d6>: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-20-e9ff310557d6>: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')

![](data:image/png;base64;base64,)

In [ ]:

def sigmoid(x):  
 """  
 Sigmoid function  
 """  
 return 1.0 / (1.0 + np.exp(-x))  
  
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 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 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()  
W\_1 = np.random.uniform(-1,1,size=(3,4))  
W\_2 = np.random.uniform(-1,1,size=(4))  
num\_iter = 5000  
learning\_rate = .001  
x\_mat = x\_mat\_full  
  
loss\_vals, accuracies = [], []  
for i in range(num\_iter):  
 ### Do a forward computation, and get the gradient  
 y\_pred, (J\_W\_1\_grad, J\_W\_2\_grad) = forward\_pass(W\_1, W\_2)  
  
 ## Update the weight matrices  
 W\_1 = W\_1 - learning\_rate\*J\_W\_1\_grad  
 W\_2 = W\_2 - learning\_rate\*J\_W\_2\_grad  
  
 ### Compute the loss and accuracy  
 curr\_loss = loss\_fn(y,y\_pred)  
 loss\_vals.append(curr\_loss)  
 acc = np.sum((y\_pred>=.5) == y)/num\_obs  
 accuracies.append(acc)  
  
 ## Print the loss and accuracy for every 200th iteration  
 if((i%200) == 0):  
 print('iteration {}, log loss is {:.4f}, accuracy is {}'.format(  
 i, curr\_loss, acc  
 ))  
plot\_loss\_accuracy(loss\_vals, accuracies)

iteration 0, log loss is 0.7097, accuracy is 0.512  
iteration 200, log loss is 0.6887, accuracy is 0.504  
iteration 400, log loss is 0.6816, accuracy is 0.56  
iteration 600, log loss is 0.6531, accuracy is 0.766  
iteration 800, log loss is 0.5639, accuracy is 0.788  
iteration 1000, log loss is 0.4624, accuracy is 0.826  
iteration 1200, log loss is 0.3851, accuracy is 0.868  
iteration 1400, log loss is 0.3370, accuracy is 0.906  
iteration 1600, log loss is 0.3069, accuracy is 0.912  
iteration 1800, log loss is 0.2855, accuracy is 0.91  
iteration 2000, log loss is 0.2684, accuracy is 0.914  
iteration 2200, log loss is 0.2543, accuracy is 0.914  
iteration 2400, log loss is 0.2425, accuracy is 0.916  
iteration 2600, log loss is 0.2323, accuracy is 0.92  
iteration 2800, log loss is 0.2219, accuracy is 0.926  
iteration 3000, log loss is 0.2093, accuracy is 0.934  
iteration 3200, log loss is 0.1979, accuracy is 0.946  
iteration 3400, log loss is 0.1889, accuracy is 0.95  
iteration 3600, log loss is 0.1816, accuracy is 0.952  
iteration 3800, log loss is 0.1756, accuracy is 0.958  
iteration 4000, log loss is 0.1704, accuracy is 0.96  
iteration 4200, log loss is 0.1659, accuracy is 0.966  
iteration 4400, log loss is 0.1619, accuracy is 0.97  
iteration 4600, log loss is 0.1583, accuracy is 0.97  
iteration 4800, log loss is 0.1550, accuracy is 0.968

![](data:image/png;base64;base64,)

References: <https://github.com/yakupkaplan/IBM---Deep-Learning-and-Reinforcement-Learning/blob/main/05c_DEMO_Backpropagation.ipynb>

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');

![](data:image/png;base64;base64,)

#### Supplementary Activity[¶](#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 [ ]:

# Define the tanh Function  
def tanh(x):  
 """Hyperbolic tangent (tanh) function"""  
 return np.tanh(x)

* The numpy function.tanh() is used to define the tanh function.

In [ ]:

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)  
y = ((np.abs(x\_mat\_full[:,0]) + np.abs(x\_mat\_full[:,1]))<1).astype(int)

* The code above is used for plotting the truth VS prediction. Copied from the procedure above

In [ ]:

# Forward pass function with tanh activation  
def forward\_pass(W1, W2):  
 global x\_mat  
 global num\_  
 global y  
 # First, compute the new predictions `y\_pred`  
 z\_2 = np.dot(x\_mat, W1)  
 a\_2 = tanh(z\_2)  
 z\_3 = np.dot(a\_2, W2)  
 y\_pred = tanh(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 = 1 - np.square(tanh(z\_2))  
 J\_W\_1\_grad = (np.dot(J\_z\_3\_grad.reshape(-1, 1), W2.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

* Depending on the number of hidden layers in the neural network, the forward\_pass function applies the activation function, which is the tanh, repeatedly while performing a forward pass. It does this by adding up all of the input values. It then uses backpropagation to convert the gradients into weights.

In [ ]:

# Loss function  
def loss\_fn(y\_true, y\_pred, epsilon = 1e-8):  
 y\_pred = np.clip(y\_pred, epsilon, 1 - epsilon)  
 return -np.mean(y\_true \* np.log(y\_pred) + (1 - y\_true) \* np.log(1 - y\_pred))

-The loss function primarily calculates the loss between y\_true and y\_pred, which are the true labels. To prevent numerical instability, the anticipated values are subsequently trimmed. Next, the cross-entrophy loss is calculated with the provided formula.

In [ ]:

x\_mat = x\_mat\_full  
num\_iter = 5000  
learning\_rate = 0.0007  
  
# Set the number of observations  
num\_obs = 500  
np.random.seed(69420)  
  
# Adjust the shapes of W1 and W2  
W\_1 = np.random.uniform(-1, 1, size=(3, 4))  
W\_2 = np.random.uniform(-1, 1, size=(4))  
  
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 = W\_1 - learning\_rate\*J\_W\_1\_grad  
 W\_2 = W\_2 - learning\_rate\*J\_W\_2\_grad  
  
 ### Compute the loss and accuracy  
 supple\_loss = loss\_fn(y, y\_pred)  
 accuracy = np.sum((y\_pred >= 0.5) == y) / num\_obs  
 loss\_vals.append(supple\_loss)  
 accuracies.append(accuracy)  
  
  
 ## Print the loss and accuracy for every 300th iteration  
 if (i % 300) == 0:  
 print('iteration {}, log loss is {:.4f}, accuracy is {}'.format(  
 i, supple\_loss, accuracy  
 ))  
  
# Plot loss and accuracy  
plt.figure(figsize=(16, 8))  
plt.subplot(1, 2, 1)  
plt.plot(loss\_vals)  
plt.title('Log Loss')  
plt.xlabel('Iterations')  
  
plt.subplot(1, 2, 2)  
plt.plot(accuracies)  
plt.title('Accuracy')  
plt.xlabel('Iterations')  
  
plt.show()

iteration 0, log loss is 9.6156, accuracy is 0.478  
iteration 300, log loss is 0.5128, accuracy is 0.746  
iteration 600, log loss is 0.5039, accuracy is 0.758  
iteration 900, log loss is 0.4186, accuracy is 0.804  
iteration 1200, log loss is 0.2727, accuracy is 0.912  
iteration 1500, log loss is 0.2427, accuracy is 0.916  
iteration 1800, log loss is 0.2334, accuracy is 0.912  
iteration 2100, log loss is 0.2294, accuracy is 0.916  
iteration 2400, log loss is 0.2276, accuracy is 0.916  
iteration 2700, log loss is 0.2267, accuracy is 0.916  
iteration 3000, log loss is 0.2263, accuracy is 0.914  
iteration 3300, log loss is 0.2260, accuracy is 0.91  
iteration 3600, log loss is 0.2259, accuracy is 0.91  
iteration 3900, log loss is 0.2259, accuracy is 0.91  
iteration 4200, log loss is 0.2260, accuracy is 0.91  
iteration 4500, log loss is 0.2261, accuracy is 0.91  
iteration 4800, log loss is 0.2262, accuracy is 0.908

![](data:image/png;base64;base64,)

* In the initial lines of code, x\_mat is assigned the value of x\_mat\_full, which contains the input data for the neural network. The variable num\_eter stores the number of epochs. Following the instructions from Hands-On 6.2, the learning rate is set to 0.0007. Random seeds are established, and the initial weights, W\_1 and W\_2, are set.
* The for loop runs according to the num\_eter variable, which specifies the number of epochs. In each iteration, it performs a forward pass in the neural network, generating predictions (y\_pred) and gradients with respect to the weights. The weights are updated using the gradients, and the loss is computed by calling the loss\_fn function, which also calculates the accuracy of the predictions. The loss and accuracy data are stored in loss\_vals and accuracies. Results are printed every 300 iterations.
* Analysis of the output shows that the log loss decreases drastically in the first epoch and remains within that range until around the 500th epoch, which seems unusual. The accuracy also increases significantly in the first few iterations, stabilizing around 0.75 from the 100th to 1000th iteration, before rising to 0.9 and staying there until the end. The sudden increase in accuracy around the 100th iteration is not yet understood.

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=7.5)  
ax.plot(x\_mat[pred0 & (y==1),0],x\_mat[pred0 & (y==1),1], 'yo', label='false negatives', markersize=7.5, alpha=.6)  
ax.set(title='Truth vs Prediction')  
ax.legend(bbox\_to\_anchor=(1, 0.8), fancybox=True, shadow=True, fontsize='large');

![](data:image/png;base64;base64,)

* The provided code plots Truth vs. Prediction. The diamond area in the middle represents the true positives, while the outside area represents the true negatives. Not all points inside the diamond are true positives; the left side contains false negatives, explaining why the model's accuracy is only 90%.

#### Conclusion[¶](#Conclusion)

* I have implemented and presented a basic neural network in this hands-on exercise. Because I had to manually carry out all neural network operations—such as establishing weights, adding up, and running activation functions—I found this hands-on activity to be challenging. Instead, I discovered that using libraries like Keras made the prior and next task simpler. With a learning rate of 0.0007, an epoch of 5000, and the activation function tanh, the aforementioned model achieves an accuracy level of 90%. I have utilised the centred diamond plot to show the true positive and negative as well as the false positive and negative when displaying the truth vs. prediction plot.

In [1]:

!pip install pandoc

Collecting pandoc  
 Downloading pandoc-2.3.tar.gz (33 kB)  
 Preparing metadata (setup.py) ... done  
Collecting plumbum (from pandoc)  
 Downloading plumbum-1.8.3-py3-none-any.whl (127 kB)  
 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 127.6/127.6 kB 2.3 MB/s eta 0:00:00  
Collecting ply (from pandoc)  
 Downloading ply-3.11-py2.py3-none-any.whl (49 kB)  
 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 49.6/49.6 kB 1.9 MB/s eta 0:00:00  
Building wheels for collected packages: pandoc  
 Building wheel for pandoc (setup.py) ... done  
 Created wheel for pandoc: filename=pandoc-2.3-py3-none-any.whl size=33263 sha256=9985df33afaa199aa8f05eca248fc3a4123d0d8a52e33b14ddf5031a8721d374  
 Stored in directory: /root/.cache/pip/wheels/76/27/c2/c26175310aadcb8741b77657a1bb49c50cc7d4cdbf9eee0005  
Successfully built pandoc  
Installing collected packages: ply, plumbum, pandoc  
Successfully installed pandoc-2.3 plumbum-1.8.3 ply-3.11

In [ ]:

!jupyter nbconvert --to html /content/CPE018HOA9\_ORB\_Feature\_Detection\_and\_Feature\_Matching\_Ramos.ipynb

In [ ]:

# Convert HTML to DOCX  
!pandoc /content/CPE018HOA9\_ORB\_Feature\_Detection\_and\_Feature\_Matching\_Ramos.html -s -o Act8.1.docx