LAB-3 AI

User ID	Device Mo	Operating	App Usage	Screen On	Battery Dr	Number of	Data Usag	Age	Gender	User Beha
1	Google Pix	Android	393	6.4	1872	67	1122	40	Male	4
2	OnePlus 9	Android	268	4.7	1331	42	944	47	Female	3
3	Xiaomi Mi	Android	154	4	761	32	322	42	Male	2
4	Google Pix	Android	239	4.8	1676	56	871	20	Male	3
5	iPhone 12	iOS	187	4.3	1367	58	988	31	Female	3
6	Google Pix	Android	99	2	940	35	564	31	Male	2
7	Samsung G	Android	350	7.3	1802	66	1054	21	Female	4
8	OnePlus 9	Android	543	11.4	2956	82	1702	31	Male	5
9	Samsung G	Android	340	7.7	2138	75	1053	42	Female	4
10	iPhone 12	iOS	424	6.6	1957	75	1301	42	Male	4
11	Google Pix	Android	53	1.4	435	17	162	34	Female	1
12	OnePlus 9	Android	215	5.5	1690	47	641	24	Male	3
13	OnePlus 9	Android	462	6.2	2303	65	1099	57	Female	4
14	Xiaomi Mi	Android	215	4.9	1662	43	857	43	Male	3
15	iPhone 12	iOS	189	5.4	1754	53	779	49	Female	3
16	Google Pix	Android	503	10.4	2571	84	2025	39	Female	5
17	OnePlus 9	Android	132	3.6	628	32	344	47	Female	2
18	iPhone 12	iOS	299	5.8	1431	41	985	44	Female	3
19	Google Pix	Android	81	1.4	558	16	297	26	Female	1
20	iPhone 12	iOS	577	8.5	2774	89	2192	29	Female	5
21	Samsung G	Android	93	2.6	681	37	302	45	Female	2
22	OnePlus 9	Android	576	11.6	2803	82	1553	43	Female	5
23	Samsung G	Android	423	6.5	2094	65	1372	23	Female	4
24	Google Pix	Android	292	5.6	1401	46	949	37	Female	3
25	OnePlus 9	Android	216	4	1711	59	748	58	Male	3
26	Samsung G	Android	91	3.4	1073	38	451	52	Male	2

User ID	ge Time (n	n Time (ho	Drain (mA	r of Apps I	Usage (MB	Age	Behavior (	Model_On	el_Samsun	lodel_Xiao	Model_iPh	ting Syste	ender_Ma
1	0.639085	0.490909	0.583426	0.606096	0.425887	0.126383	4	0	0	0	0	0	1
2	0.419014	0.336364	0.382386	-0.32244	0.351566	0.709506	3	1	0	0	0	0	0
3	0.21831	0.272727	0.170569	-0.69386	0.091858	0.292989	2	0	0	1	0	0	1
4	0.367958	0.345455	0.510591	0.19754	0.321086	-1.53968	3	0	0	0	0	0	1
5	0.276408	0.3	0.395764	0.271823	0.369937	-0.62335	3	0	0	0	1	1	0
6	0.121479	0.090909	0.237087	-0.58243	0.192902	-0.62335	2	0	0	0	0	0	1
7	0.56338	0.572727	0.557414	0.568954	0.397495	-1.45638	4	0	1	0	0	0	0
8	0.903169	0.945455	0.98625	1.163218	0.668058	-0.62335	5	1	0	0	0	0	1
9	0.545775	0.609091	0.682274	0.903228	0.397077	0.292989	4	0	1	0	0	0	0
10	0.693662	0.509091	0.615013	0.903228	0.500626	0.292989	4	0	0	0	1	1	1
11	0.040493	0.036364	0.049424	-1.25098	0.025052	-0.37344	1	0	0	0	0	0	0
12	0.325704	0.409091	0.515793	-0.13673	0.225052	-1.20647	3	1	0	0	0	0	1
13	0.760563	0.472727	0.74359	0.531813	0.416284	1.542538	4	1	0	0	0	0	0
14	0.325704	0.354545	0.505388	-0.2853	0.31524	0.376293	3	0	0	1	0	0	1
15	0.27993	0.4	0.539576	0.086115	0.282672	0.876112	3	0	0	0	1	1	0
16	0.832746	0.854545	0.843181	1.237501	0.802923	0.04308	5	0	0	0	0	0	0
17	0.179577	0.236364	0.121145	-0.69386	0.101044	0.709506	2	1	0	0	0	0	0
18	0.473592	0.436364	0.419547	-0.35958	0.368685	0.459596	3	0	0	0	1	1	0
19	0.089789	0.036364	0.095132	-1.28812	0.08142	-1.03986	1	0	0	0	0	0	0
20	0.963028	0.681818	0.918618	1.423209	0.872651	-0.78995	5	0	0	0	1	1	0
21	0.110915	0.145455	0.14084	-0.50815	0.083507	0.542899	2	0	1	0	0	0	0
22	0.961268	0.963636	0.929394	1.163218	0.605846	0.376293	5	1	0	0	0	0	0
23	0.691901	0.5	0.665923	0.531813	0.530271	-1.28977	4	0	1	0	0	0	0
24	0.461268	0.418182	0.408398	-0.17388	0.353653	-0.12353	3	0	0	0	0	0	0
25	0.327465	0.272727	0.523597	0.308964	0.269729	1.625842	3	1	0	0	0	0	1
26	0.107394	0.218182	0.286511	-0.47101	0.14572	1.126022	2	0	1	0	0	0	1
27	0.728873	0.6	0.769231	0.977511	0.375783	-0.78995	4	0	0	0	1	1	1
28	0.848592	0.863636	0.78298	1.423209	0.625052	-0.45674	5	0	0	0	0	0	1
29	0.742958	0.527273	0.774805	0.977511	0.565344	1.375932	4	1	0	0	0	0	0

```python
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# Load the preprocessed data
data = pd.read_excel("/path_to/preprocessed_user_behavior_data.xlsx")
# Separate features (X) and labels (Y)
X = data.drop(columns=['User Behavior Class'])  # assuming 'User Behavior Class' is the output label
Y = data['User Behavior Class']
# Split into training and testing datasets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)
# Define the neural network architecture
model = Sequential()
model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))  # Input layer and first hidden layer
model.add(Dense(32, activation='relu'))  # Second hidden layer
model.add(Dense(1, activation='softmax'))  # Output layer for classification
# Compile the model
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# Train the model
model.fit(X_train, Y_train, epochs=50, batch_size=32, validation_data=(X_test, Y_test))
# Predict on the test set
Y_pred = model.predict(X_test)
# Convert probabilities to class predictions
Y_pred_classes = np.argmax(Y_pred, axis=1)
# Print some predictions along with actual labels
for i in range(10):  # Show the first 10 predictions
    print(f"Predicted: {Y_pred_classes[i]}, Actual: {Y_test.values[i]}")
# Evaluate the model on test data
loss, accuracy = model.evaluate(X_test, Y_test)
print(f"Test Accuracy: {accuracy}")
```

```
        "dW1": dW1,
        "db1": db1,
        "dW2": dW2,
        "db2": db2
    }
    return grads
def update_parameters(parameters, grads, learning_rate):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]
    W1 = W1 - learning_rate*dW1
    b1 = b1 - learning_rate*db1
    W2 = W2 - learning_rate*dW2
    b2 = b2 - learning_rate*db2
    new_parameters = {
        "W1": W1,
        "W2": W2,
        "b1" : b1,
        "b2" : b2
    }
    return new_parameters
def model(X, Y, n_x, n_h, n_y, num_of_iters, learning_rate):
    parameters = initialize_parameters(n_x, n_h, n_y)
    for i in range(0, num_of_iters+1):
        a2, cache = forward_prop(X, parameters)

        cost = calculate_cost(a2, Y)

        grads = backward_prop(X, Y, cache, parameters)

        parameters = update_parameters(parameters, grads, learning_rate)

        if(i%100 == 0):
            print('Cost after iteration# {:d}: {:f}'.format(i, cost))
    return parameters
def predict(X, parameters):
    a2, cache = forward_prop(X, parameters)
    yhat = a2
    yhat = np.squeeze(yhat)
```

```python
import numpy as np
def sigmoid(z):
    return 1/(1 + np.exp(-z))
def initialize_parameters(n_x, n_h, n_y):
    W1 = np.random.randn(n_h, n_x)
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h)
    b2 = np.zeros((n_y, 1))
    parameters = {
        "W1": W1,
        "b1" : b1,
        "W2": W2,
        "b2" : b2
    }
    return parameters
def forward_prop(X, parameters):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
    cache = {
        "A1": A1,
        "A2": A2
    }
    return A2, cache
def calculate_cost(A2, Y):
    cost = -np.sum(np.multiply(Y, np.log(A2)) +  np.multiply(1-Y, np.log(1-A2)))/m
    cost = np.squeeze(cost)
    return cost
def backward_prop(X, Y, cache, parameters):
    A1 = cache["A1"]
    A2 = cache["A2"]
    W2 = parameters["W2"]
    dZ2 = A2 - Y
    dW2 = np.dot(dZ2, A1.T)/m
    db2 = np.sum(dZ2, axis=1, keepdims=True)/m
    dZ1 = np.multiply(np.dot(W2.T, dZ2), 1-np.power(A1, 2))
    dW1 = np.dot(dZ1, X.T)/m
    db1 = np.sum(dZ1, axis=1, keepdims=True)/m
```

```python
def predict(X, parameters):
    a2, cache = forward_prop(X, parameters)
    yhat = a2
    yhat = np.squeeze(yhat)
    if(yhat >= 0.5):
        y_predict = 1
    else:
        y_predict = 0

    return y_predict
np.random.seed(2)
# The 4 training examples by columns
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]])
# The outputs of the XOR for every example in X
Y = np.array([[1, 0, 0, 1]])
# No. of training examples
m = X.shape[1]
# Set the hyperparameters
n_x = 2     #No. of neurons in first layer
n_h = 2     #No. of neurons in hidden layer
n_y = 1     #No. of neurons in output layer
num_of_iters = 1000
learning_rate = 0.3
trained_parameters = model(X, Y, n_x, n_h, n_y, num_of_iters, learning_rate)
# Test 2X1 vector to calculate the XOR of its elements.
# Try (0, 0), (0, 1), (1, 0), (1, 1)
X_test = np.array([[1], [1]])
y_predict = predict(X_test, trained_parameters)
print('Neural Network prediction for example ({:d}, {:d}) is {:d}'.format(
    X_test[0][0], X_test[1][0], y_predict))
```

```python
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append(v)

    def depth_limited_dfs(self, node, goal, depth):
        if depth == 0 and node == goal:
            return True
        if depth > 0:
            for neighbor in self.graph.get(node, []):
                if self.depth_limited_dfs(neighbor, goal, depth - 1):
                    return True
        return False

    def iddfs(self, start, goal):
        depth = 0
        while True:
            print(f"Depth: {depth}")
            if self.depth_limited_dfs(start, goal, depth):
                return True
            depth += 1

# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge('A', 'B')
    g.add_edge('A', 'C')
    g.add_edge('B', 'D')
    g.add_edge('B', 'E')
    g.add_edge('C', 'F')
    g.add_edge('C', 'G')

    start_node = 'A'
    goal_node = 'E'

    if g.iddfs(start_node, goal_node):
        print(f"Goal {goal_node} found!")
    else:
        print(f"Goal {goal_node} not found.")
```

Depth: 0

Depth: 1

Depth: 2

Goal E found!