```
# HW6 - creating a Neural Network
```

```
import numpy as np
import pandas as pd
import tensorflow as tf

from typing import Tuple
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
data = pd.read_csv(
    "https://raw.githubusercontent.com/changyaochen/MECE4520/master/"
    "data/breast_cancer.csv")
data
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactnes
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	
...	...	...	...	...	...	...	...	
564	926424	M	21.56	22.39	142.00	1479.0	0.11100	
565	926682	M	20.13	28.25	131.20	1261.0	0.09780	
566	926954	M	16.60	28.08	108.30	858.1	0.08455	
567	927241	M	20.60	29.33	140.10	1265.0	0.11780	
568	92751	B	7.76	24.54	47.92	181.0	0.05263	

569 rows × 32 columns

```
# encode the diagnosis = M as 1, and diagnosis = B as 0 - treat as a binary outcome (dependent variable)
data["label"] = data["diagnosis"].apply(lambda x: 0 if x == "B" else 1)
```

```
# use 3 features - namely smoothness_mean, texture_mean, and perimeter_mean - to build a Neural Network (NN)
```

```
# define a list of features that will be used to train the NN
features = [
    # "radius_mean",
    "texture_mean",
    "perimeter_mean",
    # "area_mean",
    "smoothness_mean",
    # "compactness_mean",
    # "concavity_mean",
    # "concave_mean",
    # "symmetry_mean",
    # "fractal_mean",
]

# the target variable for the classification task is specified as "label"
label = "label"

# train_test_split - function splits the data into training and testing sets
# 'X_raw' = features & 'Y' = labels for training
X_raw, X_raw_test, Y, Y_test = train_test_split(
    data[features].values, data[label].values, test_size=0.2, random_state=42
)

# standardize the input using StandardScalar function - ensures each feature has a mean of 0 & standard devi
# input features = 'X_raw' & 'X_raw_test'
scaler = StandardScaler()
scaler.fit(X_raw)
X = scaler.transform(X_raw)             # standardization
X_test = scaler.transform(X_raw_test)   # standardization
```

```
# formatting - reshape labels 'Y' & 'Y_test' to have a single column
Y = Y.reshape((-1, 1))
Y_test = Y_test.reshape((-1, 1))
```

```
# need one hidden layer together with the final layer(output layer)
# hidden layer - 5 neurons (units) and use ReLU as the activation function
# final layer - 1 neuron (unit) and use the sigmoid function as the activations
```

```
# building the neural network
model = tf.keras.Sequential([                                                    # creates a linear stack
    tf.keras.layers.Dense(5, activation='relu', input_shape=(len(features),)),      # (input) hidden layer
    tf.keras.layers.Dense(1, activation='sigmoid')                                  # (output) final layer
])

# compile the model - sets the model up for training
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])   # adam - optimization al
                                                                                    # loss function for bina
                                                                                    # display the accuracy o

# train the model - trains the model on the input data (X) with corresponding labels (Y)
model.fit(X, Y, epochs=50, batch_size=32, validation_data=(X_test, Y_test))        # model will iterate 50
                                                                                    # model will use 32 samp
                                                                                    # validation_data uses t
```

```
Epoch 22/50
15/15 [==============================] - 0s 5ms/step - loss: 0.6562 - accuracy: 0.7385 - val_loss: 0.653
Epoch 23/50
15/15 [==============================] - 0s 5ms/step - loss: 0.6487 - accuracy: 0.7626 - val_loss: 0.645
Epoch 24/50
15/15 [==============================] - 0s 6ms/step - loss: 0.6413 - accuracy: 0.7868 - val_loss: 0.636
Epoch 25/50
15/15 [==============================] - 0s 4ms/step - loss: 0.6337 - accuracy: 0.7934 - val_loss: 0.628
Epoch 26/50
```

```
15/15 [==============================] - 0s 5ms/step - loss: 0.6261 - accuracy: 0.7978 - val_loss: 0.620
Epoch 27/50
15/15 [==============================] - 0s 4ms/step - loss: 0.6186 - accuracy: 0.8000 - val_loss: 0.612
Epoch 28/50
15/15 [==============================] - 0s 4ms/step - loss: 0.6112 - accuracy: 0.8154 - val_loss: 0.603
Epoch 29/50
15/15 [==============================] - 0s 4ms/step - loss: 0.6039 - accuracy: 0.8242 - val_loss: 0.595
Epoch 30/50
15/15 [==============================] - 0s 5ms/step - loss: 0.5967 - accuracy: 0.8242 - val_loss: 0.587
Epoch 31/50
15/15 [==============================] - 0s 6ms/step - loss: 0.5895 - accuracy: 0.8330 - val_loss: 0.580
Epoch 32/50
15/15 [==============================] - 0s 4ms/step - loss: 0.5823 - accuracy: 0.8462 - val_loss: 0.572
Epoch 33/50
15/15 [==============================] - 0s 4ms/step - loss: 0.5753 - accuracy: 0.8571 - val_loss: 0.564
Epoch 34/50
15/15 [==============================] - 0s 5ms/step - loss: 0.5683 - accuracy: 0.8571 - val_loss: 0.557
Epoch 35/50
15/15 [==============================] - 0s 6ms/step - loss: 0.5614 - accuracy: 0.8571 - val_loss: 0.549
Epoch 36/50
15/15 [==============================] - 0s 5ms/step - loss: 0.5545 - accuracy: 0.8615 - val_loss: 0.542
Epoch 37/50
15/15 [==============================] - 0s 6ms/step - loss: 0.5479 - accuracy: 0.8615 - val_loss: 0.535
Epoch 38/50
15/15 [==============================] - 0s 4ms/step - loss: 0.5411 - accuracy: 0.8681 - val_loss: 0.528
Epoch 39/50
15/15 [==============================] - 0s 5ms/step - loss: 0.5347 - accuracy: 0.8725 - val_loss: 0.521
Epoch 40/50
15/15 [==============================] - 0s 4ms/step - loss: 0.5281 - accuracy: 0.8835 - val_loss: 0.515
Epoch 41/50
15/15 [==============================] - 0s 5ms/step - loss: 0.5216 - accuracy: 0.8835 - val_loss: 0.508
Epoch 42/50
15/15 [==============================] - 0s 4ms/step - loss: 0.5150 - accuracy: 0.8857 - val_loss: 0.501
Epoch 43/50
15/15 [==============================] - 0s 5ms/step - loss: 0.5085 - accuracy: 0.8879 - val_loss: 0.495
Epoch 44/50
15/15 [==============================] - 0s 5ms/step - loss: 0.5020 - accuracy: 0.8945 - val_loss: 0.488
```

```
Epoch 45/50
15/15 [==============================] - 0s 5ms/step - loss: 0.4955 - accuracy: 0.8967 - val_loss: 0.481
Epoch 46/50
15/15 [==============================] - 0s 5ms/step - loss: 0.4888 - accuracy: 0.8923 - val_loss: 0.474
Epoch 47/50
15/15 [==============================] - 0s 5ms/step - loss: 0.4819 - accuracy: 0.8923 - val_loss: 0.467
Epoch 48/50
15/15 [==============================] - 0s 5ms/step - loss: 0.4749 - accuracy: 0.8945 - val_loss: 0.460
Epoch 49/50
15/15 [==============================] - 0s 4ms/step - loss: 0.4674 - accuracy: 0.8989 - val_loss: 0.452
Epoch 50/50
15/15 [==============================] - 0s 6ms/step - loss: 0.4597 - accuracy: 0.9011 - val_loss: 0.444
<keras.src.callbacks.History at 0x7fb3842b91b0>
```

```
# how many parameters are there in this NN?
# formula to calculate parameters in a dense layer:  Number of Parameters = (Input Size(=len(features)) + 1)
# calculate the number of parameters for each layer, then add together to get total number number of paramet
```

```
features      # features to use
```

```
['texture_mean', 'perimeter_mean', 'smoothness_mean']
```

```
len(features) # value corresponding to len(features)
```

```
3
```

```
# count the number of parameters in each layer
def count_parameters(layer):
    return layer.count_params()

# calculate and print the number of parameters for each layer
for i, layer in enumerate(model.layers):
    print(f"Parameters in Layer {i+1}: {count_parameters(layer)}")

# calculate and print the total number of parameters in the model
total_parameters = sum(count_parameters(layer) for layer in model.layers)
print(f"Total Parameters: {total_parameters}")
```

```
Parameters in Layer 1: 20
Parameters in Layer 2: 6
Total Parameters: 26
```

```
# calculating parameters using professor's github example - check above

# number of units/neurons in each layer
layer1 = 5
layer2 = 1

# input size - the length of the features
input_size = len(features)

# given weight and bias matrices
W_1_init = np.random.normal(size=(input_size, layer1))
b_1_init = np.random.normal(size=(1, layer1))
W_2_init = np.random.normal(size=(layer1, layer2))
b_2_init = np.random.normal(size=(1, layer2))

# calculate the number of parameters for each layer
layer1_parameters = np.prod(W_1_init.shape) + np.prod(b_1_init.shape)
layer2_parameters = np.prod(W_2_init.shape) + np.prod(b_2_init.shape)

# total number of parameters for the whole NN
total_parameters = layer1_parameters + layer2_parameters

# Print the results
print(f"Parameters in Layer 1: {layer1_parameters}")
print(f"Parameters in Layer 2: {layer2_parameters}")
print(f"Total Parameters: {total_parameters}")
```

```
Parameters in Layer 1: 20
Parameters in Layer 2: 6
Total Parameters: 26
```

```
# print the given weight and bias matrices for clarification
```

```
print("Initial Weight and Bias Matrices: \n")
print(f"\nWight matrix for the First (input/hidden) Layer is:\n {W_1_init}")
print(f"\nShape of W_1_init is: {W_1_init.shape}")
print(f"\n\nBias matrix for the First (input/hidden) Layer is:\n {b_1_init}")
print(f"\nShape of b_1_init is: {b_1_init.shape}")
print(f"\n\nWeight matrix for the Second (output/final) Layer is:\n {W_2_init}")
print(f"\nShape of W_2_init is: {W_2_init.shape}")
print(f"\n\nBias matrix for the Second (output/final) Layer is:\n {b_2_init}")
print(f"\nShape of b_2_init is: {b_2_init.shape}")
```

```
Initial Weight and Bias Matrices:


Wight matrix for the First (input/hidden) Layer is:
 [[ 0.64439681 -2.08026255  1.11976114  0.76383797 -0.09116977]
 [-1.01440616  0.70768036  0.9491325  -0.58714403 -0.02217379]
 [-0.45851533  0.18071205 -1.4811526   0.37075141 -2.47270042]]

Shape of W_1_init is: (3, 5)


Bias matrix for the First (input/hidden) Layer is:
 [[-1.89757757  0.75490643 -0.09723325 -0.84532679 -0.20675158]]

Shape of b_1_init is: (1, 5)


Weight matrix for the Second (output/final) Layer is:
 [[ 0.05597036]
 [ 1.05997012]
 [-0.29890712]
 [-1.24458316]
 [ 0.35308053]]

Shape of W_2_init is: (5, 1)


Bias matrix for the Second (output/final) Layer is:
 [[0.00753352]]

Shape of b_2_init is: (1, 1)
```

```
W_1_init = np.ones((input_size, layer1))
b_1_init = 0.1 * np.ones((1, layer1))
W_2_init = np.ones((layer1, layer2))
b_2_init = 0.1 * np.ones((1, layer2))
```

```
# both matrices W_1 and W_2 are filled w/ the value 1, and B_1 and B_2 are filled w/ the value 0.1 - notatio
# given matrices and values
W_1 = W_1_init
W_2 = W_2_init
b_1 = b_1_init
b_2 = b_2_init
# transpose each matrix - following notation from slides
W_1_T = np.transpose(W_1)
W_2_T = np.transpose(W_2)
b_1_T = np.transpose(b_1)
b_2_T = np.transpose(b_2)

# print
print("Final Matrices:")

print(f"\nW_1 is:\n {W_1}")
print(f"\nb_1 is:\n {b_1}")
print(f"\nW_2 is:\n {W_2}")
print(f"\nb_2 is:\n {b_2}")
print("\nShape of each matrix before being transposed: ")
print(f"Shape of W_1 is {W_1.shape}")
print(f"Shape of b_1 is {b_1.shape}")
print(f"Shape of W_2 is {W_2.shape}")
print(f"Shape of b_2 is {b_2.shape}")

print("\n\nTransposed Matrices: ")
print(f"\nW_1_T is:\n {W_1_T}")
```

```
print(f"\nb_1_T is:\n {b_1_T}")
print(f"\nW_2_T is:\n {W_2_T}")
print(f"\nb_2_T is:\n {b_2_T}")
print("\nShape of each matrix after its transposed: ")
print(f"Shape of W_1_T is {W_1_T.shape}")
print(f"Shape of b_1_T is {b_1_T.shape}")
print(f"Shape of W_2_T is {W_2_T.shape}")
print(f"Shape of b_2_T is {b_2_T.shape}")
```

```
Final Matrices:

W_1 is:
 [[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]

b_1 is:
 [[0.1 0.1 0.1 0.1 0.1]]

W_2 is:
 [[1.]
 [1.]
 [1.]
 [1.]
 [1.]]

b_2 is:
 [[0.1]]

Shape of each matrix before being transposed:
Shape of W_1 is (3, 5)
Shape of b_1 is (1, 5)
Shape of W_2 is (5, 1)
Shape of b_2 is (1, 1)
```

```
Transposed Matrices:

W_1_T is:
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]

b_1_T is:
 [[0.1]
 [0.1]
 [0.1]
 [0.1]
 [0.1]]

W_2_T is:
 [[1. 1. 1. 1. 1.]]

b_2_T is:
 [[0.1]]

Shape of each matrix after its transposed:
Shape of W_1_T is (5, 3)
Shape of b_1_T is (5, 1)
Shape of W_2_T is (1, 5)
Shape of b_2_T is (1, 1)
```

```
# input data --> input features are X
X = np.random.rand(5, 3)  # data with 5 neurons/units and 3 features

# standardize input - for each feature subtract the mean and divide by standard deviation before the forward
mean_X = np.mean(X, axis=0)
std_X = np.std(X, axis=0)
X_standardized = (X - mean_X) / std_X

# one forward propagation
hidden_layer_output = np.dot(X_standardized, W_1_init) + b_1_init     # output of the first/hidden layer
hidden_layer_activation = np.maximum(0, hidden_layer_output)          # applies ReLU activation function to
output_layer = np.dot(hidden_layer_activation, W_2_init) + b_2_init   # output of the final layer
output = 1 / (1 + np.exp(-output_layer))                              # applies Sigmoid activation function

# calculate the average prediction for the entire dataset
average_prediction = np.mean(output)

# Print the result
print(f"\nAverage Prediction after one forward propagation: {average_prediction}")
```

```
Average Prediction after one forward propagation: 0.7149538145179137
```

```
# use logloss as the error metric
# what is the value of this error metric after one forward propogation
# note: logloss is taken as the average over all n samples

# the logloss (aka, cross-entropy) measures how close the prediction results to the probability interpretati
# a perfect model has a logloss of 0
# logloss = - 1/n(sum((y_i(ln(p_i))) + (1 - y_i)ln(1 - p_1)))
        # n = number of samples
        # y_i = true label of the ith sample
        # p_i = predicted probabilities
```

```
data
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactnes
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	
...	...	...	...	...	...	...	...	
564	926424	M	21.56	22.39	142.00	1479.0	0.11100	
565	926682	M	20.13	28.25	131.20	1261.0	0.09780	
566	926954	M	16.60	28.08	108.30	858.1	0.08455	
567	927241	M	20.60	29.33	140.10	1265.0	0.11780	
568	92751	B	7.76	24.54	47.92	181.0	0.05263	

569 rows × 33 columns

```
# encode the diagnosis = M as 1, and diagnosis = B as 0 - treat as a binary outcome (dependent variable)
data["label"] = data["diagnosis"].apply(lambda X: 0 if X == "B" else 1)
y = data["label"]
print(data["label"])
```

```
0      1
1      1
2      1
3      1
4      1
      ..
564    1
565    1
566    1
567    1
568    0
Name: label, Length: 569, dtype: int64
```

```
print("Shape of the label matrix:", data["label"].shape)
```

```
Shape of the label matrix: (569,)
```

```
# reshape matrices - ensure output is within valid range (avoiding log(0) issues)
output = np.array(output).flatten() # predicted probabilities - y_pred
y = np.array(y).flatten()           # true labels - y_true

# calculate logloss
# logloss formula = - 1/n(sum((y_i(ln(p_i))) + (1 - y_i)ln(1 - p_i)))
def calculate_logloss(Y, output):
  epsilon = 1e-15
  output = np.clip(output, epsilon, 1 - epsilon)

  loss = - np.mean(y * np.log(output) + (1 - y) * np.log(1 - output))
  return loss

logloss = calculate_logloss(y, output)

# print
print(f"The logloss after one forward propogation is: {logloss}")
```

```
The logloss after one forward propogation is: 0.7826307182735618
```