Neural Networks for Binary Classification

In this assignment, you will use a neural network to detect breast cancer.

1 - Packages

First, let's run the cell below to import all the packages that you will need during this assignment.

- numpy is the fundamental package for scientific computing with Python.
- tensorflow a popular platform for machine learning.

In [1]:
```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

Tensorflow and Keras
Tensorflow is a machine learning package developed by Google. In 2019, Google integrated Keras into Tensorflow and released Tensorflow 2.0. Keras is a framework developed independently by François Chollet that creates a simple, layer-centric interface to Tensorflow. This course will be using the Keras interface.

2 - Neural Networks

2.1 Problem Statement

In this assignment, you will use a neural network to detect breast cancer. This is a binary classification task. This assignment will show you how the methods you have learned can be used for this classification task.

2.2 Dataset

You will start by loading the dataset for this task. The data set contains 569 training examples with 30 features. The first and second columns are the ID numbers and Diagnosis (M = malignant, B = benign), respectively. Columns 3-32 represent the 30 features.

- Write a `load_data()` function that loads the data into variables `X` and `y`

```
    - Each training example becomes a single row in our data matrix `X`.
    - This gives us a 569 x 30 matrix `X` where every row is a training example.
```

$$X = \begin{pmatrix} - - - (x^{(1)}) - - - \\ - - - (x^{(2)}) - - - \\ \vdots \\ - - - (x^{(m)}) - - - \end{pmatrix}$$

- The second part of the training set is a 569 x 1 dimensional vector `y` that contains labels for the training set
  - `y = 0` if the diagnosis is benign, `y = 1` if the diagnosis is malignant.

In [2]:
```
def load_data(filename):
    """
    Loads and formats data from the WDBC dataset

    Args:
    filename : relative path for the file that holds the data

    Returns:
    X : (ndarray Shape (m,n)) data, m examples by n features
    y : (array_like Shape (m,)) outputs, 1 == malignant, 0 == benign
    """
    # Load the data from the file
    data = np.loadtxt(filename, dtype=str, delimiter=',')

    # Store the 30 features from each example into a 2D matrix and convert the type to
    X = np.array(data[:,2:32])
    X = X.astype(float)

    # Store the outputs for each example and set each 'M' to a 1 and each 'B' to a 0
    y_tmp = np.array(data[:,1])
    numRows = y_tmp.shape[0]
    y = np.zeros((numRows,1))

    for i in range(0, numRows):
        # For each output, set to 1 if 'M' or 0 if 'B'
        if y_tmp[i] == 'M':
            y[i] = 1
        else:
            y[i] = 0

    # Return data and outputs
    return X, y
```

In [3]:
```
# Load dataset
X, y = load_data("./data/wdbc.data")
```

2.2.1 View the variables

Let's get more familiar with your dataset.

- A good place to start is to print out each variable and see what it contains.

The code below prints elements of the variables `X` and `y`.

In [4]:
```
print ('The first element of X is: ', X[0])
print ('The first element of y is: ', y[0])
```

```
The first element of X is:  [1.799e+01 1.038e+01 1.228e+02 1.001e+03 1.184e-01 2.776e-
01 3.001e-01
 1.471e-01 2.419e-01 7.871e-02 1.095e+00 9.053e-01 8.589e+00 1.534e+02
 6.399e-03 4.904e-02 5.373e-02 1.587e-02 3.003e-02 6.193e-03 2.538e+01
 1.733e+01 1.846e+02 2.019e+03 1.622e-01 6.656e-01 7.119e-01 2.654e-01
 4.601e-01 1.189e-01]
The first element of y is:  [1.]
```

2.2.2 Check the dimensions of your variables

Another way to get familiar with your data is to view its dimensions. Please print the shape of `X` and `y` and see how many training examples you have in your dataset.

In [5]:
```
print ('The shape of X is: ' + str(X.shape))
print ('The shape of y is: ' + str(y.shape))
```

```
The shape of X is: (569, 30)
The shape of y is: (569, 1)
```

Expected Output

```
The shape of X is: (569, 30)
The shape of y is: (569, 1)
```

More complex algorithms like neural network often need more training examples. Tile/copy our data to increase the training set size and reduce the number of training epochs.

In [6]:
```
X = np.tile(X,(100,1))
y= np.tile(y,(100,1))

print(X.shape, y.shape)
```

```
(56900, 30) (56900, 1)
```

2.3 Model representation

The neural network you will use in this assignment is shown in the figure below.

- This has three dense layers with sigmoid activations.

- The parameters have dimensions that are sized for a neural network with 25 units in layer 1, 15 units in layer 2 and 1 output unit in layer 3.
    - Recall that the dimensions of these parameters are determined as follows:
        - If network has s_{in} units in a layer and s_{out} units in the next layer, then
            - W will be of dimension $s_{in} \times s_{out}$.
            - b will a vector with s_{out} elements
    - Therefore, the shapes of `W`, and `b`, are
        - layer1: The shape of `W1` is (30, 25) and the shape of `b1` is (25,)
        - layer2: The shape of `W2` is (25, 15) and the shape of `b2` is: (15,)
        - layer3: The shape of `W3` is (15, 1) and the shape of `b3` is: (1,)

> **Note:** The bias vector `b` could be represented as a 1-D (n,) or 2-D (1,n) array. Tensorflow utilizes a 1-D representation and this assignment will maintain that convention.

2.4 Tensorflow Model Implementation

Tensorflow models are built layer by layer. A layer's input dimensions (s_{in} above) are calculated for you. You specify a layer's *output dimensions* and this determines the next layer's input dimension. The input dimension of the first layer is derived from the size of the input data specified in the `model.fit` statement.

> **Note:** It is also possible to add an input layer that specifies the input dimension of the first layer. For example:
> `tf.keras.Input(shape=(30,)),    #specify input shape`
> We will include that here to illuminate some model sizing.

Exercise 1

Below, using Keras Sequential model and Dense Layer with a sigmoid activation to construct the network described above.

In [7]:
```
# UNQ_C1
# GRADED CELL: Sequential model

model = Sequential(
    [
        tf.keras.Input(shape=(30,)),   #specify input size
        ### START CODE HERE ###
        tf.keras.layers.Dense(25, activation='sigmoid'),
        tf.keras.layers.Dense(15, activation='sigmoid'),
        tf.keras.layers.Dense(1, activation='sigmoid')
        ### END CODE HERE ###
    ], name = "my_model"
)
```

In [8]:
```
model.summary()
```

```
Model: "my_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 25)                775

 dense_1 (Dense)             (None, 15)                390

 dense_2 (Dense)             (None, 1)                 16

=================================================================
Total params: 1,181
Trainable params: 1,181
Non-trainable params: 0
_________________________________________________________________
```

▶ Expected Output (Click to Expand)

The parameter counts shown in the summary correspond to the number of elements in the weight and bias arrays as shown below.

In [9]:
```
L1_num_params = 30 * 25 + 25  # W1 parameters  + b1 parameters
L2_num_params = 25 * 15 + 15   # W2 parameters  + b2 parameters
L3_num_params = 15 * 1 + 1     # W3 parameters  + b3 parameters
print("L1 params = ", L1_num_params, ", L2 params = ", L2_num_params, ",  L3 params =
```

```
L1 params =  775 , L2 params =  390 ,  L3 params =  16
```

Let's further examine the weights to verify that tensorflow produced the same dimensions as we calculated above.

In [10]:
```
[layer1, layer2, layer3] = model.layers
```

In [11]:
```
#### Examine Weights shapes
W1,b1 = layer1.get_weights()
W2,b2 = layer2.get_weights()
W3,b3 = layer3.get_weights()
print(f"W1 shape = {W1.shape}, b1 shape = {b1.shape}")
print(f"W2 shape = {W2.shape}, b2 shape = {b2.shape}")
print(f"W3 shape = {W3.shape}, b3 shape = {b3.shape}")
```

```
W1 shape = (30, 25), b1 shape = (25,)
W2 shape = (25, 15), b2 shape = (15,)
W3 shape = (15, 1), b3 shape = (1,)
```

Expected Output

```
W1 shape = (30, 25), b1 shape = (25,)
W2 shape = (25, 15), b2 shape = (15,)
W3 shape = (15, 1), b3 shape = (1,)
```

`xx.get_weights` returns a NumPy array. One can also access the weights directly in their tensor form. Note the shape of the tensors in the final layer.

In [12]:
```
print(model.layers[2].weights)
```

```
[<tf.Variable 'dense_2/kernel:0' shape=(15, 1) dtype=float32, numpy=
array([[ 0.07709551],
       [-0.08410531],
       [-0.569871  ],
       [ 0.30786932],
       [ 0.59655577],
       [-0.47036856],
       [ 0.25555074],
       [ 0.09999037],
       [ 0.14107138],
       [-0.38288683],
       [ 0.45650536],
       [-0.58292234],
       [-0.3103798 ],
       [-0.10932976],
       [ 0.25482398]], dtype=float32)>, <tf.Variable 'dense_2/bias:0' shape=(1,) dtype
=float32, numpy=array([0.], dtype=float32)>]
```

The following code will define a loss function and run gradient descent to fit the weights of the model to the training data. This will be explained in more detail in the following week.

In [13]:
```
model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(0.001),
)

model.fit(
    X,y,
```

```
    epochs=10
)
```

```
Epoch 1/10
1779/1779 [==============================] - 4s 2ms/step - loss: 0.2708
Epoch 2/10
1779/1779 [==============================] - 3s 2ms/step - loss: 0.1847
Epoch 3/10
1779/1779 [==============================] - 3s 2ms/step - loss: 0.1515
Epoch 4/10
1779/1779 [==============================] - 3s 2ms/step - loss: 0.1167
Epoch 5/10
1779/1779 [==============================] - 3s 2ms/step - loss: 0.0957
Epoch 6/10
1779/1779 [==============================] - 3s 2ms/step - loss: 0.0839
Epoch 7/10
1779/1779 [==============================] - 3s 2ms/step - loss: 0.0879
Epoch 8/10
1779/1779 [==============================] - 3s 2ms/step - loss: 0.0794
Epoch 9/10
1779/1779 [==============================] - 3s 2ms/step - loss: 0.0781
Epoch 10/10
1779/1779 [==============================] - 3s 2ms/step - loss: 0.0748
```

Out[13]:
```
<keras.callbacks.History at 0x2625c9aa910>
```

To run the model on an example to make a prediction, use Keras `predict`. The input to `predict` is an array so the single example is reshaped to be two dimensional.

In [14]:
```
prediction = model.predict(X[0].reshape(1,30))  # a malignant case
print(f" predicting - a malignant case: {prediction}")
prediction = model.predict(X[20].reshape(1,30))  # a benigh case
print(f" predicting - a benigh case:  {prediction}")
```

```
1/1 [==============================] - 0s 82ms/step
 predicting - a malignant case: [[0.9997163]]
1/1 [==============================] - 0s 41ms/step
 predicting - a benigh case:  [[0.01232616]]
```

The output of the model is interpreted as a probability. In the first example above, the diagnosis is one (i.e., malignant). The model predicts the probability that the patient has breast cancel is closer to one. In the second example, the diagnosis is zero (i.e., benigh). The model predicts the probability that the patient has breast cancer is closer to zero. As in the case of logistic regression, the probability is compared to a threshold to make a final prediction.

In [15]:
```
if prediction >= 0.5:
    yhat = 1
else:
    yhat = 0
print(f"prediction after threshold: {yhat}")
```

```
prediction after threshold: 0
```

In [16]:
```
p = model.predict(X)
print('Train Accuracy: %f'%(np.mean((p>=0.5) == y) * 100))
```

```
1779/1779 [==============================] - 3s 1ms/step
Train Accuracy: 95.782074
```

2.5 NumPy Model Implementation (Forward Prop in NumPy)

As described in lecture, it is possible to build your own dense layer using NumPy. This can then be utilized to build a multi-layer neural network.

Exercise 2

Below, build a dense layer subroutine. You need to utilize a for loop to visit each unit (`j`) in the layer and perform the dot product of the weights for that unit and sum the bias for the unit to form `z` . An activation function `g(z)` is then applied to that result. This section will not utilize the matrix operations discussed in the lectures.

In [17]:
```
# UNQ_C2
# GRADED FUNCTION: my_dense

def my_dense(a_in, W, b, g):
    """
    Computes dense layer
    Args:
      a_in (ndarray (n, )) : Data, 1 example
      W    (ndarray (n,j)) : Weight matrix, n features per unit, j units
      b    (ndarray (j, )) : bias vector, j units
      g    activation function (e.g. sigmoid, relu..)
    Returns
      a_out (ndarray (j,))  : j units
    """
    units = W.shape[1]
    a_out = np.zeros(units)
### START CODE HERE ###
    for j in range(0, units):
        a_out[j] = g( np.dot(W[:, j], a_in) + b[j] )

### END CODE HERE ###
    return(a_out)
```

In [18]:
```
# GRADED FUNCTION: sigmoid
def sigmoid(z):
    """
    Compute the sigmoid of z

    Args:
        z (ndarray): A scalar, numpy array of any size.

    Returns:
        g (ndarray): sigmoid(z), with the same shape as z

    """

    z = np.clip( z, -500, 500 )           # protect against overflow
    ### START CODE HERE ###
    g = 1 / (1 + np.exp(-1*z))


    ### END SOLUTION ###
```

```
    return g
```

In [19]:
```
# Quick Check
x_tst = 0.1*np.arange(1,3,1).reshape(2,)  # (1 examples, 2 features)
W_tst = 0.1*np.arange(1,7,1).reshape(2,3) # (2 input features, 3 output features)
b_tst = 0.1*np.arange(1,4,1).reshape(3,)  # (3 features)
A_tst = my_dense(x_tst, W_tst, b_tst, sigmoid)
print(A_tst)
```

```
[0.54735762 0.57932425 0.61063923]
```

Expected Output

```
[0.54735762 0.57932425 0.61063923]
```

Finish the following cell to build a three-layer neural network utilizing the `my_dense` subroutine above.

In [20]:
```
def my_sequential(x, W1, b1, W2, b2, W3, b3):

    a1 = my_dense(x, W1, b1, sigmoid)
    a2 = my_dense(a1, W2, b2, sigmoid)
    a3 = my_dense(a2, W3, b3, sigmoid)

    return(a3)
```

We can copy trained weights and biases from Tensorflow.

In [21]:
```
W1_tmp,b1_tmp = layer1.get_weights()
W2_tmp,b2_tmp = layer2.get_weights()
W3_tmp,b3_tmp = layer3.get_weights()
```

In [22]:
```
# make predictions
prediction = my_sequential(X[0], W1_tmp, b1_tmp, W2_tmp, b2_tmp, W3_tmp, b3_tmp) # pre
if prediction >= 0.5:
    yhat = 1
else:
    yhat = 0
print( "yhat = ", yhat, " label= ", y[0,0])
prediction = my_sequential(X[20], W1_tmp, b1_tmp, W2_tmp, b2_tmp, W3_tmp, b3_tmp) # pr
if prediction >= 0.5:
    yhat = 1
else:
    yhat = 0
print( "yhat = ", yhat, " label= ", y[20,0])
```

```
yhat =  1  label=  1.0
yhat =  0  label=  0.0
```

Expected Output

```
yhat =  1  label=  1.0
yhat =  0  label=  0.0
```

2.8 NumPy Broadcasting Tutorial

In our lecture, we discussed how to use matrix multiplication to implement a dense layer where $\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{b}$. This implemenation utilized NumPy broadcasting to expand the vector $\mathbf{b}$. If you are not familiar with NumPy Broadcasting, this short tutorial is provided.

$\mathbf{X}\mathbf{W}$ is a matrix-matrix operation with dimensions $(m, j_1)(j_1, j_2)$ which results in a matrix with dimension (m, j_2). To that, we add a vector $\mathbf{b}$ with dimension $(1, j_2)$. $\mathbf{b}$ must be expanded to be a (m, j_2) matrix for this element-wise operation to make sense. This expansion is accomplished automatically for you by NumPy broadcasting.

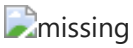
Broadcasting applies to element-wise operations.
Its basic operation is to 'stretch' a smaller dimension by replicating elements to match a larger dimension.

More specifically: When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are compatible when

- they are equal, or
- one of them is 1

If these conditions are not met, a ValueError: operands could not be broadcast together exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the size that is not 1 along each axis of the inputs.

Here are some examples:



Calculating Broadcast Result shape

For each of the following examples, try to guess the size of the result before running the example.

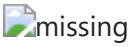
In [23]:
```
a = np.array([1,2,3]).reshape(-1,1)  #(3,1)
b = 5
print(f"(a + b).shape: {(a + b).shape}, \na + b = \n{a + b}")
```

```
(a + b).shape: (3, 1),
a + b =
[[6]
 [7]
 [8]]
```

Note that this applies to all element-wise operations:

In [24]:
```
a = np.array([1,2,3]).reshape(-1,1)  #(3,1)
b = 5
print(f"(a * b).shape: {(a * b).shape}, \na * b = \n{a * b}")
```

```
(a * b).shape: (3, 1),
a * b =
[[ 5]
 [10]
 [15]]
```

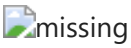


Row-Column Element-Wise Operations

In [25]:
```
a = np.array([1,2,3,4]).reshape(-1,1)
b = np.array([1,2,3]).reshape(1,-1)
print(a)
print(b)
print(f"(a + b).shape: {(a + b).shape}, \na + b = \n{a + b}")
```

```
[[1]
 [2]
 [3]
 [4]]
[[1 2 3]]
(a + b).shape: (4, 3),
a + b =
[[2 3 4]
 [3 4 5]
 [4 5 6]
 [5 6 7]]
```

This is the scenario in $\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{b}$. Adding a 1-D vector b to a (m,j) matrix.



Matrix + 1-D Vector

In [ ]:
```
```