Practice Lab: Neural Networks for Handwritten Digit Recognition, Binary

In this exercise, you will use a neural network to recognize the hand-written digits zero and one.

Outline

*NOTE: To prevent errors from the autograder, you are not allowed to edit or delete non-graded cells in this notebook . Please also refrain from adding any new cells. **Once you have passed this assignment** and want to experiment with any of the non-graded code, you may follow the instructions at the bottom of this notebook.*

1 - Packages

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

- [numpy (https://numpy.org/)](https://numpy.org/) is the fundamental package for scientific computing with Python.
- [matplotlib (http://matplotlib.org)](http://matplotlib.org) is a popular library to plot graphs in Python.
- [tensorflow (https://www.tensorflow.org/)](https://www.tensorflow.org/) a popular platform for machine learning.

```
In [8]:    1  import numpy as np
           2  import tensorflow as tf
           3  from tensorflow.keras.models import Sequential
           4  from tensorflow.keras.layers import Dense
           5  import matplotlib.pyplot as plt
           6  from autils import *
           7  %matplotlib inline
           8
           9  import logging
          10  logging.getLogger("tensorflow").setLevel(logging.ERROR)
```

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

In Course 1, you implemented logistic regression. This was extended to handle non-linear boundaries using polynomial regression. For even more complex scenarios such as image recognition, neural networks are preferred.

2.1 Problem Statement

In this exercise, you will use a neural network to recognize two handwritten digits, zero and one. This is a binary classification task. Automated handwritten digit recognition is widely used today - from recognizing zip codes (postal codes) on mail envelopes to recognizing amounts written on bank checks. You will extend this network to recognize all 10 digits (0-9) in a future assignment.

This exercise 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 `load_data()` function shown below loads the data into variables `X` and `y`

- The data set contains 1000 training examples of handwritten digits [1], here limited to zero and one.
    - Each training example is a 20-pixel x 20-pixel grayscale image of the digit.
        - Each pixel is represented by a floating-point number indicating the grayscale intensity at that location.
        - The 20 by 20 grid of pixels is "unrolled" into a 400-dimensional vector.
        - Each training example becomes a single row in our data matrix `X`.
        - This gives us a 1000 x 400 matrix `X` where every row is a training example of a handwritten digit image.

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

- The second part of the training set is a 1000 x 1 dimensional vector y that contains labels for the training set
    - y = 0 if the image is of the digit 0 , y = 1 if the image is of the digit 1 .

In [9]:   1   `# Load dataset`

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 [10]:

```
The first element of X is:  [ 0.00000000e+00  0.00000000e+00  0.00000000e+
00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  8.56059680e-06
 1.94035948e-06 -7.37438725e-04 -8.13403799e-03 -1.86104473e-02
```

In [11]:   1   `print ('The first element of y is: ', y[0,0])`

```
The first element of y is:  0
The last 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 [12]:    1  print ('The shape of X is: ' + str(X.shape))
```

```
The shape of X is: (1000, 400)
The shape of y is: (1000, 1)
```

2.2.3 Visualizing the Data

You will begin by visualizing a subset of the training set.

- In the cell below, the code randomly selects 64 rows from X , maps each row back to a 20 pixel by 20 pixel grayscale image and displays the images together.
- The label for each image is displayed above the image

```
In [13]:  1  import warnings
          2  warnings.simplefilter(action='ignore', category=FutureWarning)
          3  # You do not need to modify anything in this cell
          4
          5  m, n = X.shape
          6
          7  fig, axes = plt.subplots(8,8, figsize=(8,8))
          8  fig.tight_layout(pad=0.1)
          9
         10  for i,ax in enumerate(axes.flat):
         11      # Select random indices
         12      random_index = np.random.randint(m)
         13
         14      # Select rows corresponding to the random indices and
         15      # reshape the image
         16      X_random_reshaped = X[random_index].reshape((20,20)).T
         17
         18      # Display the image
         19      ax.imshow(X_random_reshaped, cmap='gray')
         20
         21      # Display the label above the image
         22      ax.set_title(y[random_index,0])
```
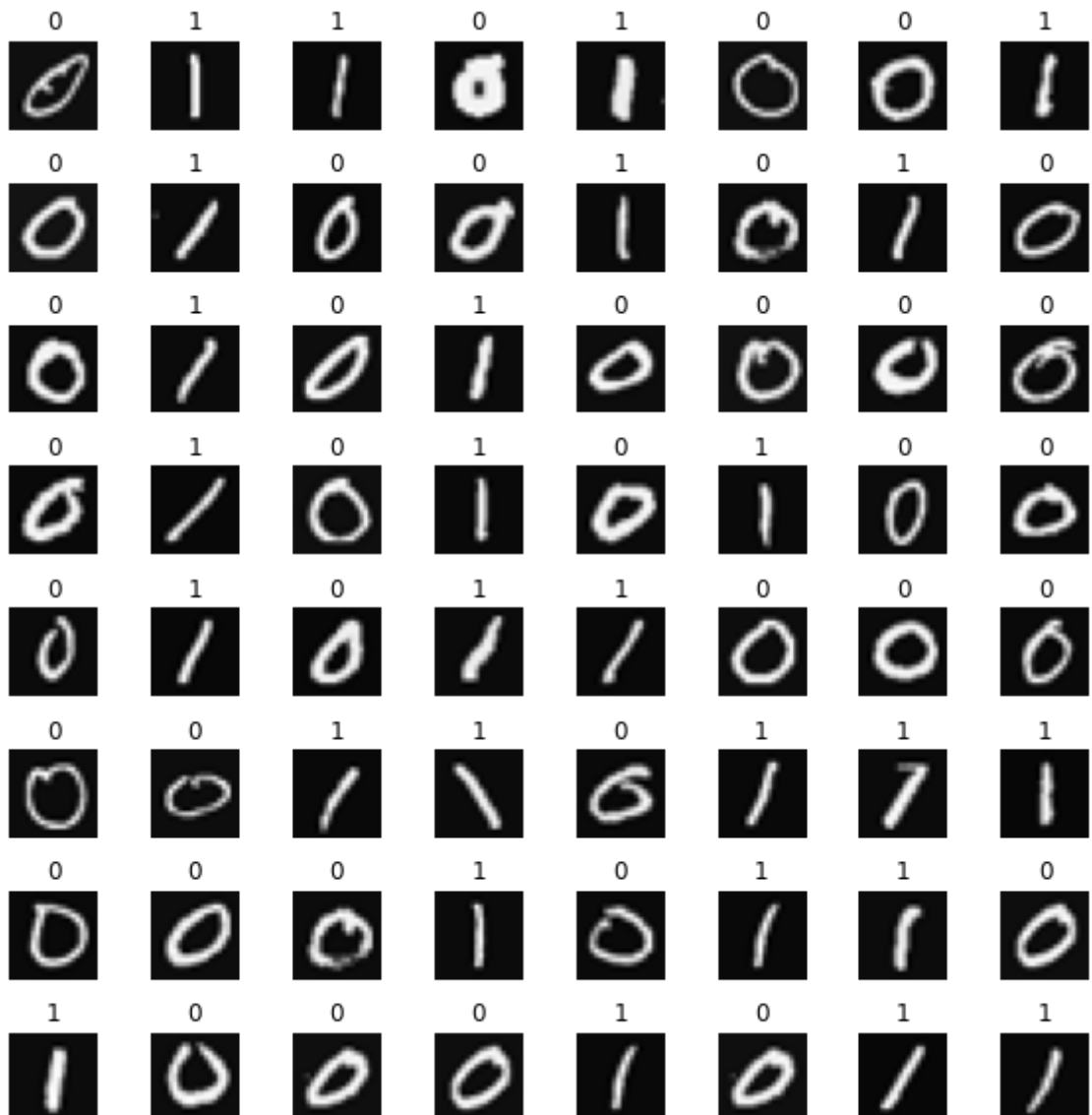


```
 1  <a name="2.3"></a>
```

- 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 (400, 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 lab 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 below.

> **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=(400,)),   #specify input shape
> We will include that here to illuminate some model sizing.

Exercise 1

Below, using Keras Sequential model (https://keras.io/guides/sequential_model/) and Dense Layer (https://keras.io/api/layers/core_layers/dense/) with a sigmoid activation to construct the network described above.

```
In [26]:   1  # UNQ_C1
           2  # GRADED CELL: Sequential model
           3
           4  model = Sequential(
           5      [
           6          tf.keras.Input(shape=(400,)),     #specify input size
           7          ### START CODE HERE ###
           8          tf.keras.layers.Dense(25,activation="sigmoid"),
           9          tf.keras.layers.Dense(15,activation="sigmoid"),
          10          tf.keras.layers.Dense(1,activation="sigmoid")
          11
          12
          13          ### END CODE HERE ###
          14      ], name = "my_model"
          15  )
```

In [27]:

```
Model: "my_model"
_____

 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 25)                10025

 dense_1 (Dense)             (None, 15)                390

 dense_2 (Dense)             (None, 1)                 16

=================================================================
Total params: 10,431
Trainable params: 10,431
Non-trainable params: 0
_____
```

Expected Output (Click to Expand)

Click for hints

```
In [28]:   1  # UNIT TESTS
           2  from public_tests import *
           3
```

```
All tests passed!
```

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

```
In [29]:   1  L1_num_params = 400 * 25 + 25   # W1 parameters  + b1 parameters
           2  L2_num_params = 25 * 15 + 15    # W2 parameters  + b2 parameters
           3  L3_num_params = 15 * 1 + 1      # W3 parameters  + b3 parameters
```

```
L1 params =  10025 , L2 params =  390 ,  L3 params =  16
```

We can examine details of the model by first extracting the layers with `model.layers` and then extracting the weights with `layerx.get_weights()` as shown below.

In [30]:

In [31]:
```
1  #### Examine Weights shapes
2  W1,b1 = layer1.get_weights()
3  W2,b2 = layer2.get_weights()
4  W3,b3 = layer3.get_weights()
5  print(f"W1 shape = {W1.shape}, b1 shape = {b1.shape}")
6  print(f"W2 shape = {W2.shape}, b2 shape = {b2.shape}")
```

```
W1 shape = (400, 25), b1 shape = (25,)
W2 shape = (25, 15), b2 shape = (15,)
W3 shape = (15, 1), b3 shape = (1,)
```

Expected Output

```
W1 shape = (400, 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 [32]:

```
[<tf.Variable 'dense_2/kernel:0' shape=(15, 1) dtype=float32, numpy=
array([[-0.51015526],
       [ 0.3259778 ],
       [ 0.36731273],
       [ 0.31615472],
       [ 0.11634964],
       [ 0.02313489],
       [ 0.25605255],
       [ 0.23651499],
       [ 0.12719369],
       [ 0.07334226],
       [-0.06693125],
       [ 0.5903911 ],
       [ 0.20416278],
       [-0.04257739],
       [ 0.45079046]], dtype=float32)>, <tf.Variable 'dense_2/bias:0' shap
e=(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 [33]:  1  model.compile(
          2      loss=tf.keras.losses.BinaryCrossentropy(),
          3      optimizer=tf.keras.optimizers.Adam(0.001),
          4  )
          5
          6  model.fit(
          7      X,y,
          8      epochs=20
```

```
Epoch 1/20
32/32 [==============================] - 0s 1ms/step - loss: 0.7657
Epoch 2/20
32/32 [==============================] - 0s 2ms/step - loss: 0.6158
Epoch 3/20
32/32 [==============================] - 0s 1ms/step - loss: 0.5069
Epoch 4/20
32/32 [==============================] - 0s 1ms/step - loss: 0.4035
Epoch 5/20
32/32 [==============================] - 0s 2ms/step - loss: 0.3204
Epoch 6/20
32/32 [==============================] - 0s 1ms/step - loss: 0.2590
Epoch 7/20
32/32 [==============================] - 0s 2ms/step - loss: 0.2143
Epoch 8/20
32/32 [==============================] - 0s 1ms/step - loss: 0.1810
Epoch 9/20
32/32 [==============================] - 0s 2ms/step - loss: 0.1558
Epoch 10/20
```

To run the model on an example to make a prediction, use Keras `predict` (https://www.tensorflow.org/api_docs/python/tf/keras/Model). The input to `predict` is an array so the single example is reshaped to be two dimensional.

```
In [34]:  1  prediction = model.predict(X[0].reshape(1,400))  # a zero
          2  print(f" predicting a zero: {prediction}")
          3  prediction = model.predict(X[500].reshape(1,400))  # a one
```

```
 predicting a zero: [[0.07123333]]
 predicting a one:  [[0.9788972]]
```

The output of the model is interpreted as a probability. In the first example above, the input is a zero. The model predicts the probability that the input is a one is nearly zero. In the second example, the input is a one. The model predicts the probability that the input is a one is nearly one. As in the case of logistic regression, the probability is compared to a threshold to make a final prediction.

```
In [35]:  1  if prediction >= 0.5:
          2      yhat = 1
          3  else:
          4      yhat = 0
```

```
prediction after threshold: 1
```

Let's compare the predictions vs the labels for a random sample of 64 digits. This takes a moment to run.
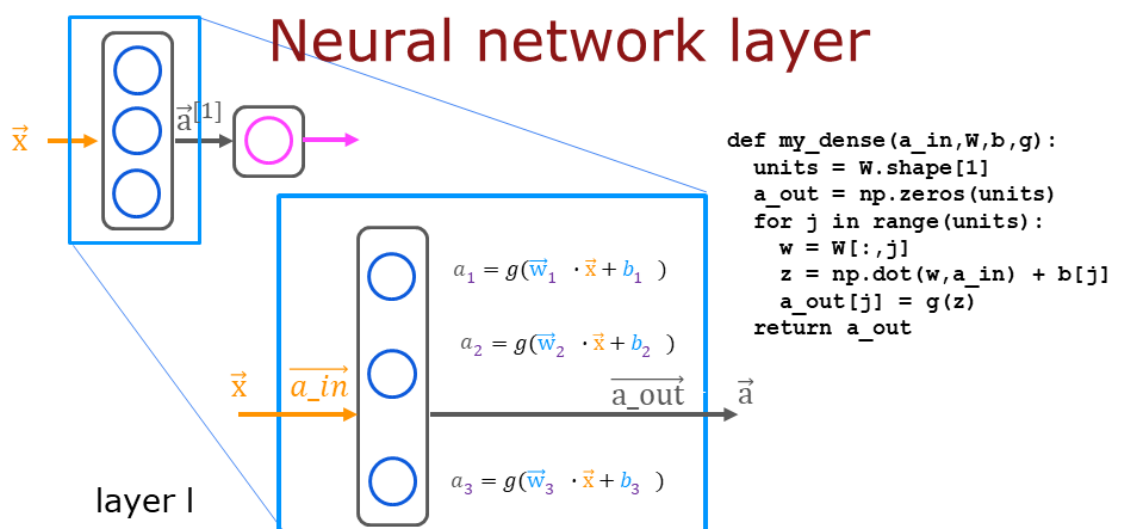
```
In [36]:  1  import warnings
          2  warnings.simplefilter(action='ignore', category=FutureWarning)
          3  # You do not need to modify anything in this cell
          4
          5  m, n = X.shape
          6
          7  fig, axes = plt.subplots(8,8, figsize=(8,8))
          8  fig.tight_layout(pad=0.1,rect=[0, 0.03, 1, 0.92]) #[left, bottom, right
          9
         10  for i,ax in enumerate(axes.flat):
         11      # Select random indices
         12      random_index = np.random.randint(m)
         13
         14      # Select rows corresponding to the random indices and
         15      # reshape the image
         16      X_random_reshaped = X[random_index].reshape((20,20)).T
         17
         18      # Display the image
         19      ax.imshow(X_random_reshaped, cmap='gray')
         20
         21      # Predict using the Neural Network
         22      prediction = model.predict(X[random_index].reshape(1,400))
         23      if prediction >= 0.5:
         24          yhat = 1
         25      else:
         26          yhat = 0
         27
         28      # Display the label above the image
         29      ax.set_title(f"{y[random_index,0]},{yhat}")
         30      ax.set_axis_off()
         31  fig.suptitle("Label, yhat", fontsize=16)
```

Label, yhat
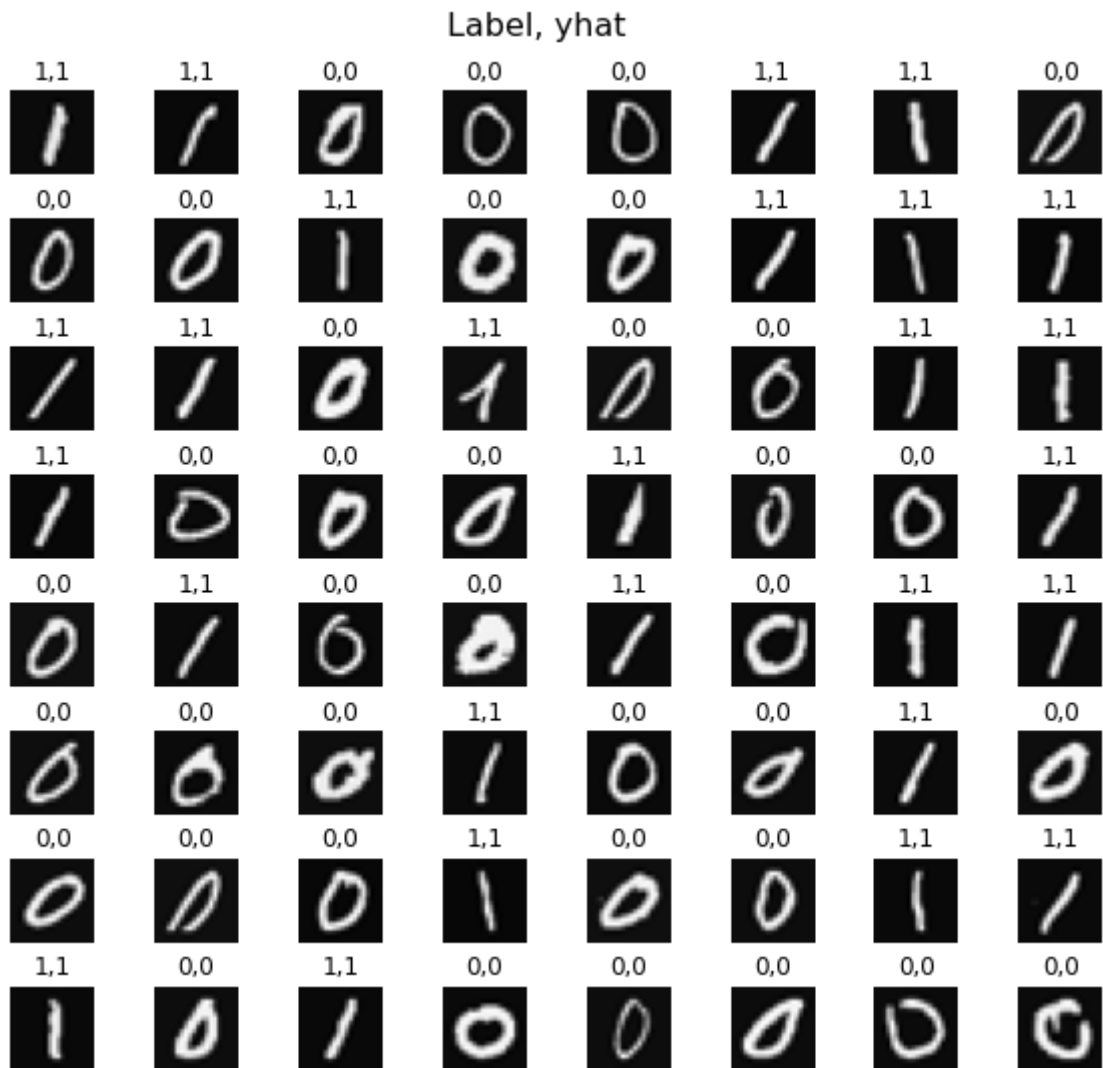
2.5 NumPy Model Implementation (Forward Prop in NumPy)

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.

Neural network layer

$$a_1 = g(\vec{w}_1 \cdot \vec{x} + b_1)$$

$$a_2 = g(\vec{w}_2 \cdot \vec{x} + b_2)$$

$$a_3 = g(\vec{w}_3 \cdot \vec{x} + b_3)$$

```
def my_dense(a_in,W,b,g):
    units = W.shape[1]
    a_out = np.zeros(units)
    for j in range(units):
        w = W[:,j]
        z = np.dot(w,a_in) + b[j]
        a_out[j] = g(z)
    return a_out
```

layer l

Exercise 2

Below, build a dense layer subroutine. The example in lecture utilized a for loop to visit each unit (j) in the layer and perform the dot product of the weights for that unit (W[:,j]) and sum the bias for the unit (b[j]) to form z . An activation function g(z) is then applied to that result. This section will not utilize some of the matrix operations described in the optional lectures. These will be explored in a later section.

```
In [37]:    1  # UNQ_C2
            2  # GRADED FUNCTION: my_dense
            3
            4  def my_dense(a_in, W, b, g):
            5      """
            6      Computes dense layer
            7      Args:
            8        a_in (ndarray (n, )) : Data, 1 example
            9        W    (ndarray (n,j)) : Weight matrix, n features per unit, j unit
           10        b    (ndarray (j, )) : bias vector, j units
           11        g      activation function (e.g. sigmoid, relu..)
           12      Returns
           13        a_out (ndarray (j,))  : j units
           14      """
           15      units = W.shape[1]
           16      a_out = np.zeros(units)
           17  ### START CODE HERE ###
           18      for j in range(units):
           19          w = W[:,j]
           20          z = np.dot(w, a_in) + b[j]
           21          a_out[j] = g(z)
           22      return(a_out)
           23
           24
           25  ### END CODE HERE ###
           26      return(a_out)
```

```
In [38]:    1  # Quick Check
            2  x_tst = 0.1*np.arange(1,3,1).reshape(2,)  # (1 examples, 2 features)
            3  W_tst = 0.1*np.arange(1,7,1).reshape(2,3) # (2 input features, 3 output
            4  b_tst = 0.1*np.arange(1,4,1).reshape(3,)  # (3 features)
            5  A_tst = my_dense(x_tst, W_tst, b_tst, sigmoid)

           [0.54735762 0.57932425 0.61063923]
```

Expected Output

```
[0.54735762 0.57932425 0.61063923]
```

Click for hints

```
In [39]:    1  # UNIT TESTS
            2
            3  test_c2(my_dense)
           All tests passed!
```

The following cell builds a three-layer neural network utilizing the `my_dense` subroutine above.

```
In [40]:    1  def my_sequential(x, W1, b1, W2, b2, W3, b3):
            2      a1 = my_dense(x,  W1, b1, sigmoid)
            3      a2 = my_dense(a1, W2, b2, sigmoid)
            4      a3 = my_dense(a2, W3, b3, sigmoid)
```

We can copy trained weights and biases from Tensorflow.

```
In [41]:    1  W1_tmp,b1_tmp = layer1.get_weights()
            2  W2_tmp,b2_tmp = layer2.get_weights()
```

```
In [42]:    1  # make predictions
            2  prediction = my_sequential(X[0], W1_tmp, b1_tmp, W2_tmp, b2_tmp, W3_tmp
            3  if prediction >= 0.5:
            4      yhat = 1
            5  else:
            6      yhat = 0
            7  print( "yhat = ", yhat, " label= ", y[0,0])
            8  prediction = my_sequential(X[500], W1_tmp, b1_tmp, W2_tmp, b2_tmp, W3_t
            9  if prediction >= 0.5:
           10      yhat = 1
           11  else:
           12      yhat = 0
```

```
yhat =  0  label=  0
yhat =  1  label=  1
```

Run the following cell to see predictions from both the Numpy model and the Tensorflow model. This takes a moment to run.

```
In [43]:   1  import warnings
           2  warnings.simplefilter(action='ignore', category=FutureWarning)
           3  # You do not need to modify anything in this cell
           4
           5  m, n = X.shape
           6
           7  fig, axes = plt.subplots(8,8, figsize=(8,8))
           8  fig.tight_layout(pad=0.1,rect=[0, 0.03, 1, 0.92]) #[left, bottom, right
           9
          10  for i,ax in enumerate(axes.flat):
          11      # Select random indices
          12      random_index = np.random.randint(m)
          13
          14      # Select rows corresponding to the random indices and
          15      # reshape the image
          16      X_random_reshaped = X[random_index].reshape((20,20)).T
          17
          18      # Display the image
          19      ax.imshow(X_random_reshaped, cmap='gray')
          20
          21      # Predict using the Neural Network implemented in Numpy
          22      my_prediction = my_sequential(X[random_index], W1_tmp, b1_tmp, W2_t
          23      my_yhat = int(my_prediction >= 0.5)
          24
          25      # Predict using the Neural Network implemented in Tensorflow
          26      tf_prediction = model.predict(X[random_index].reshape(1,400))
          27      tf_yhat = int(tf_prediction >= 0.5)
          28
          29      # Display the label above the image
          30      ax.set_title(f"{y[random_index,0]},{tf_yhat},{my_yhat}")
          31      ax.set_axis_off()
          32  fig.suptitle("Label, yhat Tensorflow, yhat Numpy", fontsize=16)
```

Label, yhat Tensorflow, yhat Numpy
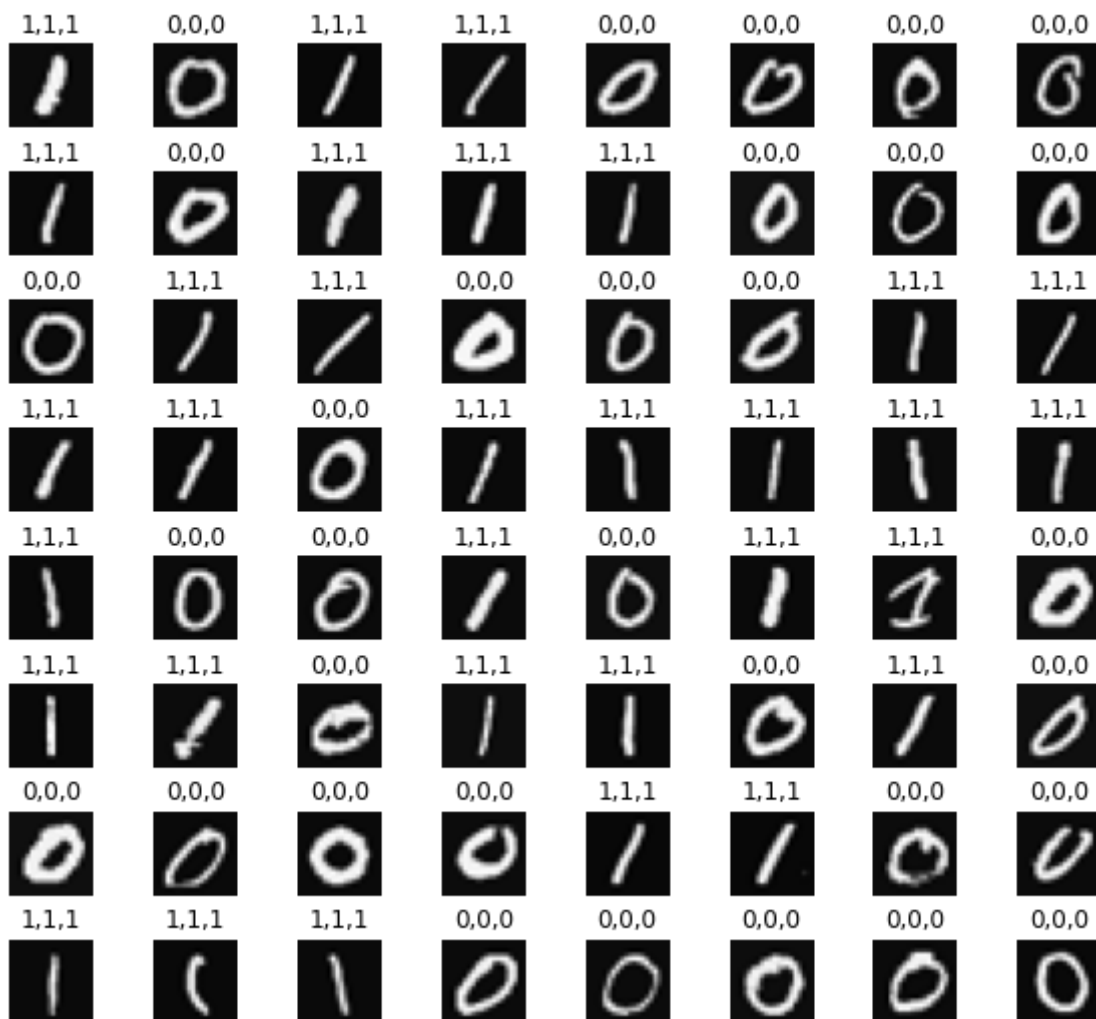
2.6 Vectorized NumPy Model Implementation (Optional)

The optional lectures described vector and matrix operations that can be used to speed the calculations. Below describes a layer operation that computes the output for all units in a layer on a given input example:

$$\mathbf{z} = \mathbf{x}_i^T \mathbf{W} \qquad [\leftarrow \quad \mathbf{x}_i^T \quad \rightarrow] \begin{bmatrix} \uparrow & \ldots & \uparrow \\ \mathbf{w}_1 & \ldots & \mathbf{w}_j \\ \downarrow & \ldots & \downarrow \end{bmatrix} = [\mathbf{x}_i^T \mathbf{w}_1 \quad \ldots \quad \mathbf{x}_i^T \mathbf{w}_j] \leftarrow example\ i$$

dimensions $\quad (1, j_{in}) \qquad (j_{in}, j_{out}) \qquad\qquad (m, j_{out})$

match

We can demonstrate this using the examples `X` and the `W1` , `b1` parameters above. We use `np.matmul` to perform the matrix multiply. Note, the dimensions of x and W must be compatible as shown in the diagram above.
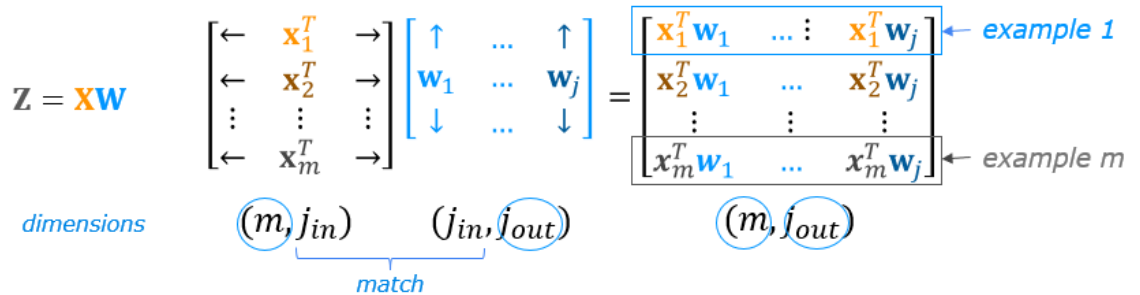
```
In [44]:  1  x = X[0].reshape(-1,1)           # column vector (400,1)
          2  z1 = np.matmul(x.T,W1) + b1      # (1,400)(400,25) = (1,25)
          3  a1 = sigmoid(z1)
```

(1, 25)

You can take this a step further and compute all the units for all examples in one Matrix-Matrix operation.

$$
\mathbf{Z} = \mathbf{XW} \quad
\begin{bmatrix}
\leftarrow & \mathbf{x}_1^T & \rightarrow \\
\leftarrow & \mathbf{x}_2^T & \rightarrow \\
\vdots & \vdots & \vdots \\
\leftarrow & \mathbf{x}_m^T & \rightarrow
\end{bmatrix}
\begin{bmatrix}
\uparrow & \cdots & \uparrow \\
\mathbf{w}_1 & \cdots & \mathbf{w}_j \\
\downarrow & \cdots & \downarrow
\end{bmatrix}
=
\begin{bmatrix}
\mathbf{x}_1^T\mathbf{w}_1 & \cdots & \vdots & \mathbf{x}_1^T\mathbf{w}_j \\
\mathbf{x}_2^T\mathbf{w}_1 & & \cdots & \mathbf{x}_2^T\mathbf{w}_j \\
& \vdots & & \vdots \\
x_m^T\mathbf{w}_1 & & \cdots & x_m^T\mathbf{w}_j
\end{bmatrix}
\begin{array}{l}
\leftarrow \textit{example 1} \\
\\
\\
\leftarrow \textit{example m}
\end{array}
$$

dimensions (m, j_{in}) (j_{in}, j_{out}) (m, j_{out})

match

The full operation is $\mathbf{Z} = \mathbf{XW} + \mathbf{b}$. This will utilize NumPy broadcasting to expand $\mathbf{b}$ to m rows. If this is unfamiliar, a short tutorial is provided at the end of the notebook.

Exercise 3

Below, compose a new `my_dense_v` subroutine that performs the layer calculations for a matrix of examples. This will utilize `np.matmul()`.

Note: This function is not graded because it is discussed in the optional lectures on vectorization. If you didn't go through them, feel free to click the hints below the expected code to see the code. You can also submit the notebook even with a blank answer here.

```
In [45]:   1  # UNQ_C3
           2  # UNGRADED FUNCTION: my_dense_v
           3
           4  def my_dense_v(A_in, W, b, g):
           5      """
           6      Computes dense layer
           7      Args:
           8        A_in (ndarray (m,n)) : Data, m examples, n features each
           9        W    (ndarray (n,j)) : Weight matrix, n features per unit, j unit
          10        b    (ndarray (1,j)) : bias vector, j units
          11        g    activation function (e.g. sigmoid, relu..)
          12      Returns
          13        A_out (tf.Tensor or ndarray (m,j)) : m examples, j units
          14      """
          15  ### START CODE HERE ###
          16
          17
          18  ### END CODE HERE ###
```

```
In [46]: 1 X_tst = 0.1*np.arange(1,9,1).reshape(4,2) # (4 examples, 2 features)
         2 W_tst = 0.1*np.arange(1,7,1).reshape(2,3) # (2 input features, 3 output
         3 b_tst = 0.1*np.arange(1,4,1).reshape(1,3) # (1,3 features)
         4 A_tst = my_dense_v(X_tst, W_tst, b_tst, sigmoid)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call las
t)
<ipython-input-46-dabeea292556> in <module>
      2 W_tst = 0.1*np.arange(1,7,1).reshape(2,3) # (2 input features, 3 o
utput features)
      3 b_tst = 0.1*np.arange(1,4,1).reshape(1,3) # (1,3 features)
----> 4 A_tst = my_dense_v(X_tst, W_tst, b_tst, sigmoid)
      5 print(A_tst)

<ipython-input-45-90ca24a90566> in my_dense_v(A_in, W, b, g)
     14     """
     15 ### START CODE HERE ###
---> 16     for i in range(units):
     17         w = W[:,i]
     18         z=np.dot(w,a_in) + b[i]

NameError: name 'units' is not defined
```

Expected Output

```
[[0.54735762 0.57932425 0.61063923]
 [0.57199613 0.61301418 0.65248946]
 [0.5962827  0.64565631 0.6921095 ]
 [0.62010643 0.67699586 0.72908792]]
```

Click for hints

```
In [ ]: 1 # UNIT TESTS
        2
```

The following cell builds a three-layer neural network utilizing the `my_dense_v` subroutine above.

```
In [47]: 1 def my_sequential_v(X, W1, b1, W2, b2, W3, b3):
         2     A1 = my_dense_v(X,  W1, b1, sigmoid)
         3     A2 = my_dense_v(A1, W2, b2, sigmoid)
         4     A3 = my_dense_v(A2, W3, b3, sigmoid)
```

We can again copy trained weights and biases from Tensorflow.

```
In [48]: 1 W1_tmp,b1_tmp = layer1.get_weights()
         2 W2_tmp,b2_tmp = layer2.get_weights()
```

Let's make a prediction with the new model. This will make a prediction on *all of the examples at once*. Note the shape of the output.

In [49]:
```python
1  Prediction = my_sequential_v(X, W1_tmp, b1_tmp, W2_tmp, b2_tmp, W3_tmp,
```

```
--------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
<ipython-input-49-1590e538921d> in <module>
----> 1 Prediction = my_sequential_v(X, W1_tmp, b1_tmp, W2_tmp, b2_tmp, W3
_tmp, b3_tmp )
      2 Prediction.shape

<ipython-input-47-ca76a4e58ea4> in my_sequential_v(X, W1, b1, W2, b2, W3,
b3)
      1 def my_sequential_v(X, W1, b1, W2, b2, W3, b3):
----> 2     A1 = my_dense_v(X,  W1, b1, sigmoid)
      3     A2 = my_dense_v(A1, W2, b2, sigmoid)
      4     A3 = my_dense_v(A2, W3, b3, sigmoid)
      5     return(A3)

<ipython-input-45-90ca24a90566> in my_dense_v(A_in, W, b, g)
     14     """
     15 ### START CODE HERE ###
---> 16     for i in range(units):
     17         w = W[:,i]
     18         z=np.dot(w,a_in) + b[i]

NameError: name 'units' is not defined
```

We'll apply a threshold of 0.5 as before, but to all predictions at once.

In [ ]:
```python
1  Yhat = (Prediction >= 0.5).astype(int)
```

Run the following cell to see predictions. This will use the predictions we just calculated above. This takes a moment to run.

```
1  import warnings
2  warnings.simplefilter(action='ignore', category=FutureWarning)
3  # You do not need to modify anything in this cell
4
5  m, n = X.shape
6
7  fig, axes = plt.subplots(8, 8, figsize=(8, 8))
8  fig.tight_layout(pad=0.1, rect=[0, 0.03, 1, 0.92]) #[left, bottom, righ
9
10 for i, ax in enumerate(axes.flat):
11     # Select random indices
12     random_index = np.random.randint(m)
13
14     # Select rows corresponding to the random indices and
15     # reshape the image
16     X_random_reshaped = X[random_index].reshape((20, 20)).T
17
18     # Display the image
19     ax.imshow(X_random_reshaped, cmap='gray')
20
21     # Display the label above the image
22     ax.set_title(f"{y[random_index,0]}, {Yhat[random_index, 0]}")
23     ax.set_axis_off()
24 fig.suptitle("Label, Yhat", fontsize=16)
```

You can see how one of the misclassified images looks.

```
1  fig = plt.figure(figsize=(1, 1))
2  errors = np.where(y != Yhat)
3  random_index = errors[0][0]
4  X_random_reshaped = X[random_index].reshape((20, 20)).T
5  plt.imshow(X_random_reshaped, cmap='gray')
6  plt.title(f"{y[random_index,0]}, {Yhat[random_index, 0]}")
7  plt.axis('off')
```

2.7 Congratulations!

You have successfully built and utilized a neural network.

2.8 NumPy Broadcasting Tutorial (Optional)

In the last example, $\mathbf{Z} = \mathbf{XW} + \mathbf{b}$ utilized NumPy broadcasting to expand the vector $\mathbf{b}$. If you are not familiar with NumPy Broadcasting, this short tutorial is provided.

$\mathbf{XW}$ 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 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 (https://NumPy.org/doc/stable/user/basics.broadcasting.html)](https://NumPy.org/doc/stable/user/basics.broadcasting.html): 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:

missing
Calculating Broadcast Result shape

The graphic below describes expanding dimensions. Note the red text below:

missing
Broadcast notionally expands arguments to match for element wise operations

The graphic above shows NumPy expanding the arguments to match before the final operation. Note that this is a notional description. The actual mechanics of NumPy operation choose the most efficient implementation.

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

```
In [ ]:   1 a = np.array([1,2,3]).reshape(-1,1)  #(3,1)
          2 b = 5
```

Note that this applies to all element-wise operations:

```
In [ ]:   1 a = np.array([1,2,3]).reshape(-1,1)  #(3,1)
          2 b = 5
```

missing
Row-Column Element-Wise Operations

```
In [ ]:   1 a = np.array([1,2,3,4]).reshape(-1,1)
          2 b = np.array([1,2,3]).reshape(1,-1)
          3 print(a)
          4 print(b)
```

This is the scenario in the dense layer you built above. Adding a 1-D vector b to a (m,j) matrix.
missing
Matrix + 1-D Vector

Please click here if you want to experiment with any of the non-graded code.