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

- 1 - Packages
- 2 - Neural Networks
  - 2.1 Problem Statement
  - 2.2 Dataset
  - 2.3 Model representation
  - 2.4 Tensorflow Model Implementation
    - Exercise 1
  - 2.5 NumPy Model Implementation (Forward Prop in NumPy)
  - 2.6 Vectorized NumPy Model Implementation (Optional)
    - Exercise 2
  - 2.7 Congratulations!
  - 2.8 NumPy Broadcasting Tutorial (Optional)

1 - Packages

Packages you will need for this assignment. You should not need any packages that you will need during this assignment.

- numpy is the fundamental package for scientific computing with Python.
- matplotlib is a popular library to plot graphs in 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
        from public_utilities import *
        %matplotlib inline
        %matplotlib inline

        import logging
        import warnings
        logging.getLogger("tensorflow").setLevel(logging.ERROR)
        tf.autograph.set_verbosity(0)
```

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 (or post codes) on mail envelopes to recognizing amounts written on bank checks. You will extend the 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 the 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.
  - Another instance of the MNIST digit recognition dataset will be used later.
  - Each pixel in represented by a floating-point number indicating the grayscale intensity at that location.
  - The 20 by 20 pixel grid 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.

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

- 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`.

[1] This is a subset of the MNIST handwritten digit dataset http://yann.lecun.com/exdb/mnist/

```
In [2]: # load dataset
        X, y = load_data()
```

2.2.1 View the variables

Let's get more familiar with your dataset.

A good first step in any new dataset is to print out each variable and see what it contains.

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

```
In [3]: print("The first element of X is: ", X[0])
```

```
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 8.56059680e-06 3.91620164e-01 8.80678941e-01
 5.96078431e-01 7.62952786e-02 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 1.37254902e-01 9.32896138e-01 9.99953909e-01
 9.99215681e-01 9.99215681e-01 9.99215681e-01 9.99215681e-01
 9.99215681e-01 9.99215681e-01 9.99215681e-01 9.99215681e-01
 9.99215681e-01 3.21948458e-01 8.67271411e-04 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 1.96832490e-03 6.27453925e-01 9.99215681e-01
 7.06832490e-01 1.68704344e-01 1.67484314e-01 1.67484314e-01
 7.06832490e-01 9.99215681e-01 9.99215681e-01 6.27453925e-01
 1.96832490e-03 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 1.80392157e-02
 6.08635097e-01 9.99215681e-01 3.04498949e-01 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 1.84852071e-03
 5.78064016e-01 9.99215681e-01 9.99215681e-01 9.30772751e-01
 2.32077340e-01 2.32077340e-01 8.50919562e-01 9.99215681e-01
 9.99215681e-01 5.78064016e-01 7.84313725e-03 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 1.37254902e-01 9.32896138e-01 9.99215681e-01
 9.99215681e-01 9.99215681e-01 9.99215681e-01 9.99215681e-01
 9.99215681e-01 9.99215681e-01 3.74117647e-01 5.53679038e-02
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
```

```
In [4]: print ("The first element of y: ", y[0,0])
        print ("The last element of y: ", y[-1,-1])
```

```
The first element of y:  0
The last element of y:  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: (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.

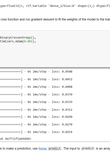
- In the cell below, the code randomly selects 64 rows from `X`, maps each row back to a 20 pixel by 20 pixel grayscale image and displays the images together.
- The label for each image is displayed above the image

```
In [6]: import warnings
        warnings.simpleformatter('ignore', category=ratarwarning)
        # this is to avoid a warning regarding to deal with axes

        m, n = X.shape

        fig, axes = utils.plot_a_photo(X, figsize=(4,4))
        for i,ax in enumerate(axes):
            # Select an index to display, randomly
            random_index = np.random.randint(m)
            # Select the corresponding row
            x_random = X[random_index, :]
            # Reshape the image
            x_random_reshaped = x_random.reshape((int(0.5*n), int(0.5*n)))

            # Display the image
            ax.imshow(x_random_reshaped, cmap='gray')

            # Display the label above the image
            ax.set_title(y[random_index,0])
            ax.set_axis_off()
```

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.
  - Recall that our inputs are pixel values of digit images.
  - Since the images are of size 20 x 20, this gives us 400 inputs



- 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 is of dimension $s_{in} \times s_{out}$.
      - b is a vector with s_{out} elements
  - Therefore, the shapes of `W`, and `b`, are
    - layer1: The shape of `W` is (400, 25) and the shape of `b` is (25,)
    - layer2: The shape of `W` is (25, 15) and the shape of `b` is: (15,)
    - layer3: The shape of `W` is (15, 1) and the shape of `b`: is (1,)

Note: The bias vector `b` could be represented as a 1-D (n,) or 2-D (1,n). 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 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.layers.InputLayer(400,),  #specify input size
            Dense(25, activation='sigmoid'),
            Dense(15, activation='sigmoid'),
            Dense(1, activation='sigmoid')
            ]                   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            )

        # END CODE HERE ###
        model.name = "my_model"
```

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

```
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 [9]: # UNQ_C1 TESTS
        from public_tests import *

        test_C1(model)
```

```
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 [10]: L1_num_params = 400 * 25  # W1 parameters  = %d
         L2_num_params = 25 * 15  # W2 parameters  = %d
         L3_num_params = 15 * 1   # W3 parameters  = %d
         print("L1 num_params = %d, L2 num_params = %d, L3 num_params = %d", % (L1_num_params, L2_num_params, L3_num_params ))
```

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

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

```
In [11]: [layer1, layer2, layer3] = model.layers

         # GRADED CODE###
         W1,b1 = layer1.get_weights()
         W2,b2 = layer2.get_weights()
         W3,b3 = layer3.get_weights()
         print("W1 shape = {W1.shape}, b1 shape = {b1.shape}")
         print("W2 shape = {W2.shape}, b2 shape = {b2.shape}")
         print("W3 shape = {W3.shape}, b3 shape = {b3.shape}")

         ### END CODE HERE ###
         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,)
```

`layer1.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([[-0.6000001],
       [-0.2000001],
       [-0.2000001],
       [-0.2000001],
       [-0.2000001],
       [-0.2000001],
       [-0.2000001],
       [-0.2000001],
       [-0.2000001],
       [-0.2000001],
       [-0.2000001],
       [-0.2000001],
       [-0.2000001],
       [-0.2000001],
       [-0.2000001]], 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=20
             )
```

```
Epoch 1/20
32/32 [==============================] - 0s 2ms/step - loss: 0.8588
Epoch 2/20
32/32 [==============================] - 0s 2ms/step - loss: 0.6452
Epoch 3/20
32/32 [==============================] - 0s 2ms/step - loss: 0.4666
Epoch 4/20
32/32 [==============================] - 0s 2ms/step - loss: 0.3179
Epoch 5/20
32/32 [==============================] - 0s 2ms/step - loss: 0.2070
Epoch 6/20
32/32 [==============================] - 0s 2ms/step - loss: 0.1312
Epoch 7/20
32/32 [==============================] - 0s 2ms/step - loss: 0.0830
Epoch 8/20
32/32 [==============================] - 0s 2ms/step - loss: 0.0537
Epoch 9/20
32/32 [==============================] - 0s 2ms/step - loss: 0.0354
Epoch 10/20
```

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[1].reshape(1,400))  # a zero
         print("prediction = %f", % (prediction))
         prediction = model.predict(X[500].reshape(1,400))  # a one
         print("prediction = %f", % (prediction))

                  prediction = zero (0.00000012)
                  prediction = one (0.99999982)
```

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. In the case of logistic regression, the probability is compared to a threshold to make a final prediction.

```
In [15]: if prediction >= 0.5:
             print("a")
             else:
             print("b")
         print("prediction after threshold: %f")

                  prediction after threshold: 1
```

Let's compare the predictions on the labels for a random sample of 64 digits. This takes a moment to run.

```
In [16]: import warnings
         warnings.simpleformatter('ignore', category=ratarwarning)
         # this is to avoid a warning regarding to deal with axes

         m, n = X.shape

         fig, axes = utils.plot_a_photo(X, figsize=(4,4))
         for i,ax in enumerate(axes):
             # Select an index to display, randomly
             random_index = np.random.randint(m)
             # Select the corresponding row
             x_random = X[random_index, :]
             # Reshape the image
             x_random_reshaped = x_random.reshape((int(0.5*n), int(0.5*n)))

             # Display the image
             ax.imshow(x_random_reshaped, cmap='gray')

             # Predict using the Neural Network
             prediction = model.predict(X[random_index].reshape(1,400))
             prediction_p = prediction * 255
             prediction_p = np.round(prediction_p)
             prediction = np.round(prediction)
             ax.set_title(prediction[0,0], fontsize=10)
             prediction_p = prediction_p[0,0].reshape((int(0.5*n), int(0.5*n)))
             # Display the label above the image
             ax.set_title(y[random_index,0]*255)
             ax.set_axis_off()
         fig.suptitle("Label, pred", fontsize=16)
         utils.dlab()
```

Label, pred

2.5 NumPy Model Implementation (Forward Prop in NumPy)

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

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 sum product of the weights times the input plus the bias for each unit. In matrix terms, this would be $z = W \cdot a + b$ where a is the output of the previous layer. This applies to that result. This section will not utilize some of the matrix operations described in the optional lecture. These will be explored in a later section.

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

         def my_dense(a_in, W, b, g):
             """
             computes dense layer
             Args:
               a_in (ndarray (s,)) : S0 is number of features
               W    (ndarray (s,j)) : shape is (number of features per unit, j units)
               b    (ndarray (j,))  : j units
               activation function (e.g. sigmoid, relu..)
             """
             units = W.shape[1]
             a_out = np.zeros(units)

             units = W.shape[1]
             for j in range(units):
             # Clear out the accumulator
             z = np.dot(W[:,j], a_in) + b
             a_out[j] = g(z)

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

```
In [18]: # quick check
         a_tst = np.array([1,2,-3,-4])
         W_tst = np.array([[.1,.2,.3,.4],[.5,.6,.7,.8],[.9,-1,-2,-3]])
         b_tst = np.array([1,-2,3,-4])
         g = lambda z: np.array([z[0]**2, z[1]**2, z[2]**2])
         a_out = my_dense(a_tst, W_tst, b_tst, g)
         print(f"a_out = {a_out}")

         [0.0377776 0.7744224 0.1444224]
```

Expected Output

```
[0.0377776 0.7744224 0.1444224]
```

Click for hints

```
In [19]: # UNQ_C2 TESTS

         test_C2(my_dense)
```

```
All tests passed!
```

The following cell builds 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)
         if prediction >= 0.5:
             print("a")
             else:
             print("b")
         print("prediction = %f", % (prediction))
         print("prediction after threshold: %f")
         prediction_p = prediction * 255
         prediction_p = np.round(prediction_p)
         prediction = np.round(prediction)
         print("prediction_p = %f", % (prediction_p))

                  prediction = zero (0.00000012)
                  prediction after threshold: 1
                  prediction_p = zero (0.000000)
```

Click for hints

```
In [23]: # UNQ TESTS

         test_C2(my_dense)
```

```
All tests passed!
```

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

```
In [24]: 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 [25]: W1_tmp,b1_tmp = layer1.get_weights()
         W2_tmp,b2_tmp = layer2.get_weights()
         W3_tmp,b3_tmp = layer3.get_weights()
```

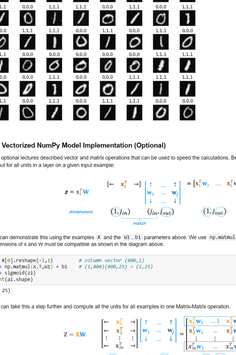
```
In [26]: # make predictions
         prediction = my_sequential(X[0], W1_tmp,b1_tmp,W2_tmp,b2_tmp,W3_tmp,b3_tmp)
         if prediction >= 0.5:
             print("a")
             else:
             print("b")
         print("prediction = %f", % (prediction))
         print("prediction after threshold: %f")
         prediction_p = prediction * 255
         prediction_p = np.round(prediction_p)
         prediction = np.round(prediction)
         print("prediction_p = %f", % (prediction_p))

                  prediction = zero (0.00000012)
                  prediction after threshold: 1
                  prediction_p = zero (0.000000)
```

Run the following cell to see predictions from both the Keras model and the NumPy model.

```
In [27]: import warnings
         warnings.simpleformatter('ignore', category=ratarwarning)
         # this is to avoid a warning regarding to deal with axes

         m, n = X.shape

         fig, axes = utils.plot_a_photo(X, figsize=(4,4))
         for i,ax in enumerate(axes):
             # Select an index to display, randomly
             random_index = np.random.randint(m)
             # Select the corresponding row
             x_random = X[random_index, :]
             # Reshape the image
             x_random_reshaped = x_random.reshape((int(0.5*n), int(0.5*n)))

             # Display the image
             ax.imshow(x_random_reshaped, cmap='gray')

             # Predict using the Neural Network
             prediction = model.predict(X[random_index].reshape(1,400))
             prediction_p = prediction * 255
             prediction_p = np.round(prediction_p)
             prediction = np.round(prediction)
             ax.set_title(prediction[0,0], fontsize=10)
             prediction_p = prediction_p[0,0].reshape((int(0.5*n), int(0.5*n)))
             # Display the label above the image
             ax.set_title(y[random_index,0]*255)
             ax.set_axis_off()
         fig.suptitle("Label, pred", fontsize=16)
         utils.dlab()
```

Label, pred

```
X_random_reshaped = X[random_index].reshape((20,20)).T

# Display the image
ax.imshow(X_random_reshaped, cmap='gray')

# Predict using the Neural Network Implemented in Numpy
my_prediction = my_sequential(X[random_index], W1_tmp, b1_tmp, W2_tmp, b2_tmp, W3_tmp, b3_tmp )
my_yhat = int(my_prediction >= 0.5)

# Predict using the Neural Network Implemented in Tensorflow
tf_prediction = model.predict(X[random_index].reshape(1,400))
tf_yhat = int(tf_prediction >= 0.5)

# Display the label above the image
ax.set_title("{}{ phat Tensorflow, phat Numpy", fontsize=16)
ax.axis('off')
fig.suptitle("Label, yhat Tensorflow, yhat Numpy", fontsize=16)
plt.show()
```



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:

$$\mathbf{z} = \mathbf{x}^T\mathbf{W} \qquad \underset{\text{dimensions}}{(1,j_{in})} \quad \underset{\text{match}}{(j_{in},j_{out})} \quad \underset{}{(1,j_{out})}$$

$$\begin{bmatrix} \leftarrow & \mathbf{x}^T & \rightarrow \end{bmatrix} \begin{bmatrix} \uparrow & & \uparrow \\ \mathbf{w}_1 & \cdots & \mathbf{w}_j \\ \downarrow & & \downarrow \end{bmatrix} = \begin{bmatrix} \mathbf{x}^T\mathbf{w}_1 & \cdots & \mathbf{x}^T\mathbf{w}_j \end{bmatrix} \leftarrow \text{example 1}$$

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 [26]:
```
x = X[0].reshape(-1,1)          # column vector (400,1)
z1 = np.matmul(x.T,W1) + b1     # (1,400)(400,25) = (1,25)
a1 = sigmoid(z1)
print(a1.shape)
```

```
(1, 25)
```

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

$$\mathbf{Z} = \mathbf{X}\mathbf{W} \qquad \underset{\text{dimensions}}{(m,j_{in})} \quad \underset{\text{match}}{(j_{in},j_{out})} \quad \underset{}{(m,j_{out})}$$

$$\begin{bmatrix} \leftarrow & \mathbf{x}^T_1 & \rightarrow \\ \leftarrow & \mathbf{x}^T_2 & \rightarrow \\ & \vdots & \\ \leftarrow & \mathbf{x}^T_m & \rightarrow \end{bmatrix} \begin{bmatrix} \uparrow & & \uparrow \\ \mathbf{w}_1 & \cdots & \mathbf{w}_j \\ \downarrow & & \downarrow \end{bmatrix} = \begin{bmatrix} \mathbf{x}^T_1\mathbf{w}_1 & \cdots & \mathbf{x}^T_1\mathbf{w}_j \\ \mathbf{x}^T_2\mathbf{w}_1 & \cdots & \mathbf{x}^T_2\mathbf{w}_j \\ \vdots & & \vdots \\ \mathbf{x}^T_m\mathbf{w}_1 & \cdots & \mathbf{x}^T_m\mathbf{w}_j \end{bmatrix} \begin{matrix} \leftarrow \text{example 1} \\ \\ \\ \leftarrow \text{example m} \end{matrix}$$

The full operation is $\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{b}$. 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 [31]:
```
# UNQ_C3
# UNGRADED FUNCTION: my_dense_v

def my_dense_v(A_in, W, b, g):
    """
    Computes dense layer
    Args:
      A_in (ndarray (m,n)) : Data, m examples, n features each
      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 (tf.Tensor or ndarray (m,)) : m examples, j units
    """
### START CODE HERE ###
    Z = np.matmul(A_in, W) + b
    A_out = g(Z)


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

In [32]:
```
W_tmp = np.array([[-0.93,0.29],[0.43,-0.26]])
b_tmp = np.array([-1.828,-0.332])
print("W_shape:", W_tmp.shape, "\n b_shape:", b_tmp.shape, "\n")
A_in = np.array([[-0.45, 0.60],[0.79, 0.24]])
A_out = my_dense_v(A_in, W_tmp, b_tmp, sigmoid)
print(A_out)
```

```
W_shape: (2, 2)
 b_shape: (2,)

[[0.57375762 0.57933425 0.61063923]
 [0.57399613 0.61383418 0.65248046]]
```

Expected Output

```
[[0.57375762 0.57933425 0.61063923]
 [0.57399613 0.61383418 0.65248046]
 [0.5962827  0.64565631 0.69219895]
 [0.62010643 0.67699586 0.72908792]]
```

Click for hints

In [33]:
```
# UNIT TESTS

test_c3(my_dense_v)
```

```
All tests passed!
```

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

In [34]:
```
def my_sequential_v(X, W1, b1, W2, b2, W3, b3):
    A1 = my_dense_v(X,  W1, b1, sigmoid)
    A2 = my_dense_v(A1, W2, b2, sigmoid)
    A3 = my_dense_v(A2, W3, b3, sigmoid)
    return(A3)
```

We can again copy trained weights and biases from Tensorflow.

In [35]:
```
W1_tmp,b1_tmp = layer1.get_weights()
W2_tmp,b2_tmp = layer2.get_weights()
W3_tmp,b3_tmp = layer3.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 [36]:
```
Prediction = my_sequential_v(X, W1_tmp, b1_tmp, W2_tmp, b2_tmp, W3_tmp, b3_tmp )
Prediction.shape
```

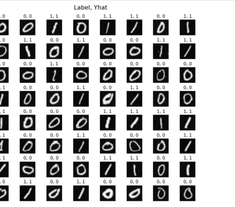
Out[36]:
```
(5000, 1)
```

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

In [37]:
```
Yhat = (Prediction >= 0.5).astype(int)
print("Prediction a zero: ",Yhat[0], "Prediction a one: ", Yhat[500])

predict a zero:  [0] predict a one:  [1]
```

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

In [38]:
```
import warnings
warnings.simplefilter('ignore', category=FutureWarning)
# You do not need to modify anything in this cell

m, n = X.shape

fig, axes = plt.subplots(8, 8, figsize=(8, 8))
fig.tight_layout(pad=0.1, rect=[0, 0.03, 1, 0.92]) #[left, bottom, right, top]

for i, ax in enumerate(axes.flat):
    # Select random indices
    random_index = np.random.randint(m)

    # Select rows corresponding to the random indices and
    # reshape the image
    X_random_reshaped = X[random_index].reshape((20, 20)).T

    # Display the image
    ax.imshow(X_random_reshaped, cmap='gray')

    # Display the label above the image
    ax.set_title("{}{ {Yhat[random_index,0]}, {Yhat[random_index, 0]}")
    ax.set_axis_off()
fig.suptitle("Label, Yhat", fontsize=16)
plt.show()
```



You can see how one of the misclassified images looks.

In [39]:
```
fig = plt.figure(figsize=(1, 1))
errors = np.where(y != Yhat)
random_index = errors[0][0]
X_random_reshaped = X[random_index].reshape((20, 20)).T
plt.imshow(X_random_reshaped, cmap='gray')
plt.title("{}{random_index,0]}, {Yhat[random_index, 0]}")
plt.axis('off')
plt.show()
```

2.7 Congratulations!

You have successfully built and utilized a neural network.

2.8 NumPy Broadcasting Tutorial (Optional)

In the last example, $\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{b}$ 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: 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

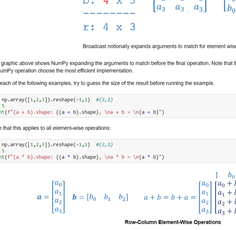
If these conditions are not met, a ValueError: operands could not be broadcast together exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the size that is not 1 along each axis of the inputs.

Here are some examples:

```
a:      m x 1     1 x n     4 x 1
b:          1     1 x n     1 x 3
-----   -----     -----     -----
result: m x 1     1 x n     4 x 3
```

Calculating Broadcast Result shape

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



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

Note that this applies to all element-wise operations:

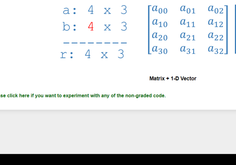
In [ ]:
```
a = np.array([1,2,3]).reshape(-1,1)  #(3,1)
b = 5
print("{^^a * b}.shape: {{a * b}.shape}, \na * b = \n{a * b}")
```

$$\boldsymbol{a} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} b_0 & b_1 & b_2 \end{bmatrix} \quad a + b = b + a = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_0 & b_1 & b_2 \\ b_0 & b_1 & b_2 \\ b_0 & b_1 & b_2 \\ b_0 & b_1 & b_2 \end{bmatrix} = \begin{bmatrix} a_0 + b_0 & a_0 + b_1 & a_0 + b_2 \\ a_1 + b_0 & a_1 + b_1 & a_1 + b_2 \\ a_2 + b_0 & a_2 + b_1 & a_2 + b_2 \\ a_3 + b_0 & a_3 + b_1 & a_3 + b_2 \end{bmatrix}$$

Row-Column Element-Wise Operations

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

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



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