# 中国矿业大学

# 计算机科学与技术学院

# 2021 级本科生课程设中期计报告

| | |
|---|---|
| 课程名称 | 机器学习应用实践课程设计 |
| 设计题目 | 线性模型与神经网络实验 |
| 开课学期 | 2023-2024 学年第 1 学期 |
| 报告时间 | 2023.10.09 |
| 学生姓名 | 杨学通 |
| 学　　号 | 08213129 |
| 班　　级 | 人工智能一班 |
| 专　　业 | 人工智能 |
| 任课教师 | 杜文亮 |

# 《机器学习应用实践课程设计》课程报告评分表

开课学期:2023-2024 学年第 1 学期

姓名:杨学通　　　　　　学号：08213129　　　　　专业班级：人工智能一班

| 序号 | 毕业要求 | 课程教学目标 | 考查方式与考查点 | 占比 | 得分 |
|---|---|---|---|---|---|
| 1 | 1.2 | **目标 1**：掌握机器学习算法的 Python 编译环境配置;掌握 Python 编程集成环境 Pychram。 | **中期检查与设计文档**<br>基于 Python 的机器学习算法编译环境搭建。 | 5% | |
| 2 | 2.2 | **目标 2**：掌握基于 numpy 库的线性模型算法搭建方法。 | **中期检查与设计文档**<br>基于 numpy 库实现线性模型的代价函数、梯度下降函数等函数的代码编写，以及测试结果。 | 12.5% | |
| 3 | 2.2 | **目标 3**：掌握基于 numpy 与 Tensorflow 库的神经网络模型算法搭建方法。 | **中期检查与设计文档**<br>基于 numpy 与 Tensorflow 库实现神经网络模型的二分类与多分类的代码编写能力，以及代码编写测试。 | 12.5% | |
| | | | **中期成绩** | 30% | |

任课教师：杜文亮

2023 年 10 月 09 日

# 实验一 线性回归

## 一、实验目的

1. 掌握本地Jupyter notebook编译环境；
2. 熟悉Python编程集成环境Pychram、vs code；
3. 掌握基于numpy库的线性回归模型搭建。 本实验代码于吴恩达老师Coursera中的机器学习专项课程。

## 二、实验内容

1. 安装并配置本地Jupyter notebook编译环境；
2. 基于numpy库实现线性回归的代价函数、梯度下降函数等。

## 三、实验结果

1. 完成本实验中的Exercise 1和Exercise 2, 直到显示All tests passed!;

## 四、实验心得(500字以内)

实验心得：通过本次实验我学会如何利用Numpy提供的向量运算函数，实现线性回归的代价函数和梯度函数。利用Numpy的库函数，可以有效的减少for循环的个数，并且向量的运算效率也高于for循环。通过分析公式，编写自己的代码，让我对公式的理解更加深入，逐渐理解了梯度下降的过程，以及其意义所在。当然，编写代码的过程中，也有犯错的时候，比如在编写梯度函数时，贪图代码简洁性'w=w*x+b'，错误的将传入的权重修改了，导致后续梯度不下降。这深深反思。在编写好梯度函数之后，便开始了进行1000多次的梯度下降过程，最终得到一条比较拟合数据集的线性预测模型。

# Practice Lab: Linear Regression

Welcome to your first practice lab! In this lab, you will implement linear regression with one variable to predict profits for a restaurant franchise.

# Outline

## 1 - Packages

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

- numpy (www.numpy.org) is the fundamental package for working with matrices in Python.
- matplotlib (http://matplotlib.org) is a famous library to plot graphs in Python.
- `utils.py` contains helper functions for this assignment. You do not need to modify code in this file.

In [50]:
```
import numpy as np
import matplotlib.pyplot as plt
from utils import *
import copy
import math
%matplotlib inline
```

## 2 - Problem Statement

Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet.

- You would like to expand your business to cities that may give your restaurant higher profits.
- The chain already has restaurants in various cities and you have data for profits and populations from the cities.
- You also have data on cities that are candidates for a new restaurant.
  - For these cities, you have the city population.

Can you use the data to help you identify which cities may potentially give your business higher profits?

## 3 - Dataset

You will start by loading the dataset for this task.

- The `load_data()` function shown below loads the data into variables `x_train` and `y_train`
  - `x_train` is the population of a city
  - `y_train` is the profit of a restaurant in that city. A negative value for profit indicates a loss.
  - Both `X_train` and `y_train` are numpy arrays.

In [51]:
```
# load the dataset
x_train, y_train = load_data()
```

**View the variables**

Before starting on any task, it is useful to get more familiar with your dataset.

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

The code below prints the variable `x_train` and the type of the variable.

In [52]:
```
# print x_train
print("Type of x_train:",type(x_train))
print("First five elements of x_train are:\n", x_train[:5])
```

```
Type of x_train: <class 'numpy.ndarray'>
First five elements of x_train are:
 [6.1101 5.5277 8.5186 7.0032 5.8598]
```

`x_train` is a numpy array that contains decimal values that are all greater than zero.

- These values represent the city population times 10,000
- For example, 6.1101 means that the population for that city is 61,101

Now, let's print `y_train`

In [53]:
```
# print y_train
print("Type of y_train:",type(y_train))
print("First five elements of y_train are:\n", y_train[:5])
```

```
Type of y_train: <class 'numpy.ndarray'>
First five elements of y_train are:
 [17.592   9.1302 13.662  11.854   6.8233]
```

Similarly, `y_train` is a numpy array that has decimal values, some negative, some positive.

- These represent your restaurant's average monthly profits in each city, in units of $10,000.
    - For example, 17.592 represents $175,920 in average monthly profits for that city.
    - -2.6807 represents -$26,807 in average monthly loss for that city.

#### Check the dimensions of your variables

Another useful way to get familiar with your data is to view its dimensions.

Please print the shape of `x_train` and `y_train` and see how many training examples you have in your dataset.

In [54]:
```
print ('The shape of x_train is:', x_train.shape)
print ('The shape of y_train is: ', y_train.shape)
print ('Number of training examples (m):', len(x_train))
```

```
The shape of x_train is: (97,)
The shape of y_train is:  (97,)
Number of training examples (m): 97
```

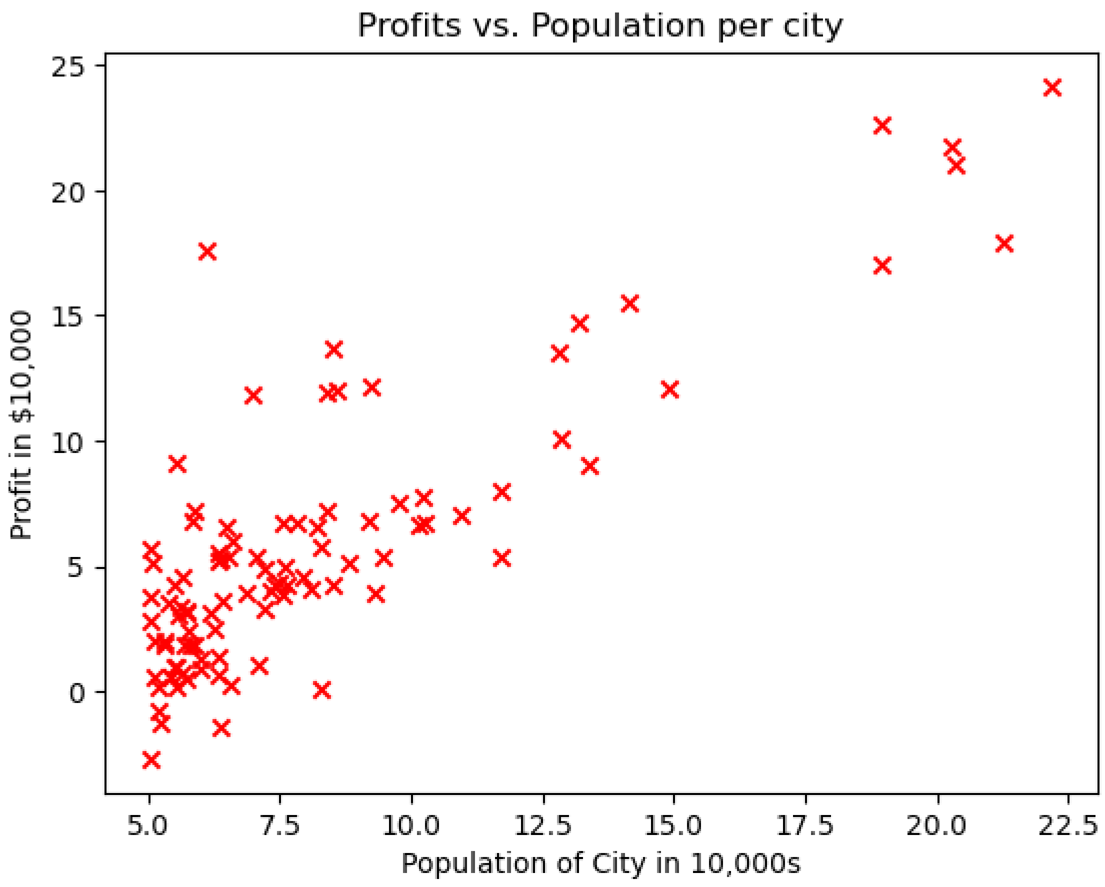
The city population array has 97 data points, and the monthly average profits also has 97 data points. These are NumPy 1D arrays.

**Visualize your data**

It is often useful to understand the data by visualizing it.

- For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population).
- Many other problems that you will encounter in real life have more than two properties (for example, population, average household income, monthly profits, monthly sales).When you have more than two properties, you can still use a scatter plot to see the relationship between each pair of properties.

In [55]:
```
# Create a scatter plot of the data. To change the markers to red "x",
# we used the 'marker' and 'c' parameters
plt.scatter(x_train, y_train, marker='x', c='r')

# Set the title
plt.title("Profits vs. Population per city")
# Set the y-axis label
plt.ylabel('Profit in $10,000')
# Set the x-axis label
plt.xlabel('Population of City in 10,000s')
plt.show()
```



Your goal is to build a linear regression model to fit this data.

- With this model, you can then input a new city's population, and have the model estimate your restaurant's potential monthly profits for that city.

## 4 - Refresher on linear regression

In this practice lab, you will fit the linear regression parameters $(w, b)$ to your dataset.

- The model function for linear regression, which is a function that maps from `x` (city population) to `y` (your restaurant's monthly profit for that city) is represented as
$$f_{w,b}(x) = wx + b$$
- To train a linear regression model, you want to find the best $(w, b)$ parameters that fit your dataset.
    - To compare how one choice of $(w, b)$ is better or worse than another choice, you can evaluate it with a cost function $J(w, b)$
        - $J$ is a function of $(w, b)$. That is, the value of the cost $J(w, b)$ depends on the value of $(w, b)$.
    - The choice of $(w, b)$ that fits your data the best is the one that has the smallest cost $J(w, b)$.
- To find the values $(w, b)$ that gets the smallest possible cost $J(w, b)$, you can use a method called **gradient descent**.
    - With each step of gradient descent, your parameters $(w, b)$ come closer to the optimal values that will achieve the lowest cost $J(w, b)$.
- The trained linear regression model can then take the input feature $x$ (city population) and output a prediction $f_{w,b}(x)$ (predicted monthly profit for a restaurant in that city).

## 5 - Compute Cost

Gradient descent involves repeated steps to adjust the value of your parameter $(w, b)$ to gradually get a smaller and smaller cost $J(w, b)$.

- At each step of gradient descent, it will be helpful for you to monitor your progress by computing the cost $J(w, b)$ as $(w, b)$ gets updated.
- In this section, you will implement a function to calculate $J(w, b)$ so that you can check the progress of your gradient descent implementation.

### Cost function

As you may recall from the lecture, for one variable, the cost function for linear regression $J(w, b)$ is defined as

$$J(w, b) = \frac{1}{2m} \sum_{i=0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)})^2$$

- You can think of $f_{w,b}(x^{(i)})$ as the model's prediction of your restaurant's profit, as opposed to $y^{(i)}$, which is the actual profit that is recorded in the data.
- $m$ is the number of training examples in the dataset

### Model prediction

- For linear regression with one variable, the prediction of the model $f_{w,b}$ for an example $x^{(i)}$ is representented as:

$$f_{w,b}(x^{(i)}) = wx^{(i)} + b$$

This is the equation for a line, with an intercept $b$ and a slope $w$

### Implementation

Please complete the `compute_cost()` function below to compute the cost $J(w, b)$.

## Exercise 1

Complete the `compute_cost` below to:

- Iterate over the training examples, and for each example, compute:
  - The prediction of the model for that example
$$f_{wb}(x^{(i)}) = wx^{(i)} + b$$
  - The cost for that example
$$cost^{(i)} = (f_{wb} - y^{(i)})^2$$
- Return the total cost over all examples
$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=0}^{m-1} cost^{(i)}$$
  - Here, $m$ is the number of training examples and $\sum$ is the summation operator

In [56]:
```
# UNQ_C1
# GRADED FUNCTION: compute_cost

def compute_cost(x, y, w, b):
    """
    Computes the cost function for linear regression.

    Args:
        x (ndarray): Shape (m,) Input to the model (Population of cities)
        y (ndarray): Shape (m,) Label (Actual profits for the cities)
        w, b (scalar): Parameters of the model

    Returns
        total_cost (float): The cost of using w,b as the parameters for linear regr
               to fit the data points in x and y
    """
    # number of training examples
    m = x.shape[0]

    # You need to return this variable correctly
    total_cost = 0.0

    ### START CODE HERE ###
    total_cost=np.dot(w*x+b-y,w*x+b-y)/2/m
    ### END CODE HERE ###

    return total_cost
```

You can check if your implementation was correct by running the following test code:

In [57]:
```
# Compute cost with some initial values for paramaters w, b
initial_w = 2
initial_b = 1

cost = compute_cost(x_train, y_train, initial_w, initial_b)
print(type(cost))
print(f'Cost at initial w: {cost:.3f}')

# Public tests
from public_tests import *
compute_cost_test(compute_cost)
```

```
<class 'numpy.float64'>
Cost at initial w: 75.203
All tests passed!
```

**Expected Output**:

**Cost at initial w: 75.203**

## 6 - Gradient descent

In this section, you will implement the gradient for parameters $w, b$ for linear regression.

As described in the lecture videos, the gradient descent algorithm is:

$$\begin{aligned}&\text{repeat until convergence: } \{\\&\quad b := b - \alpha\frac{\partial J(w,b)}{\partial b}\\&\quad w := w - \alpha\frac{\partial J(w,b)}{\partial w}\\&\}\end{aligned} \tag{1}$$

where, parameters $w, b$ are both updated simultaniously and where

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{w,b}(x^{(i)}) - y^{(i)}) \tag{2}$$

$$\frac{\partial J(w,b)}{\partial w} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{w,b}(x^{(i)}) - y^{(i)})x^{(i)} \tag{3}$$

- m is the number of training examples in the dataset
- $f_{w,b}(x^{(i)})$ is the model's prediction, while $y^{(i)}$, is the target value

You will implement a function called `compute_gradient` which calculates $\frac{\partial J(w)}{\partial w}$, $\frac{\partial J(w)}{\partial b}$

### Exercise 2

Please complete the `compute_gradient` function to:

- Iterate over the training examples, and for each example, compute:

- The prediction of the model for that example
$$f_{wb}(x^{(i)}) = wx^{(i)} + b$$
- The gradient for the parameters $w, b$ from that example
$$\frac{\partial J(w,b)}{\partial b}^{(i)} = (f_{w,b}(x^{(i)}) - y^{(i)})$$
$$\frac{\partial J(w,b)}{\partial w}^{(i)} = (f_{w,b}(x^{(i)}) - y^{(i)})x^{(i)}$$

- Return the total gradient update from all the examples
$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} \frac{\partial J(w,b)}{\partial b}^{(i)}$$
$$\frac{\partial J(w,b)}{\partial w} = \frac{1}{m} \sum_{i=0}^{m-1} \frac{\partial J(w,b)}{\partial w}^{(i)}$$
  - Here, $m$ is the number of training examples and $\sum$ is the summation operator

In [58]:
```
# UNQ_C2
# GRADED FUNCTION: compute_gradient
def compute_gradient(x, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      x (ndarray): Shape (m,) Input to the model (Population of cities)
      y (ndarray): Shape (m,) Label (Actual profits for the cities)
      w, b (scalar): Parameters of the model
    Returns
      dj_dw (scalar): The gradient of the cost w.r.t. the parameters w
      dj_db (scalar): The gradient of the cost w.r.t. the parameter b
     """


    # Number of training examples
    m = x.shape[0]

    # You need to return the following variables correctly
    dj_dw = 0
    dj_db = 0

    ### START CODE HERE ###
    dj_dw=np.dot(w*x+b-y,x)/m
    dj_db=np.sum(w*x+b-y)/m
    ### END CODE HERE ###

    return dj_dw, dj_db
```

Run the cells below to check your implementation of the `compute_gradient` function with two different initializations of the parameters $w$,$b$.

In [59]:
```
# Compute and display gradient with w initialized to zeroes
initial_w = 0
initial_b = 0

tmp_dj_dw, tmp_dj_db = compute_gradient(x_train, y_train, initial_w, initial_b)
print('Gradient at initial w, b (zeros):', tmp_dj_dw, tmp_dj_db)

compute_gradient_test(compute_gradient)
```

```
Gradient at initial w, b (zeros): -65.32884974555671 -5.839135051546393
Using X with shape (4, 1)
All tests passed!
```

Now let's run the gradient descent algorithm implemented above on our dataset.

**Expected Output**:

| | |
|---|---|
| **Gradient at initial , b (zeros)** | -65.32884975 -5.83913505154639 |

In [60]:
```
# Compute and display cost and gradient with non-zero w
test_w = 0.2
test_b = 0.2
tmp_dj_dw, tmp_dj_db = compute_gradient(x_train, y_train, test_w, test_b)

print('Gradient at test w, b:', tmp_dj_dw, tmp_dj_db)
```

```
Gradient at test w, b: -47.41610118114432 -4.007175051546392
```

**Expected Output**:

| | |
|---|---|
| **Gradient at test w** | -47.41610118 -4.007175051546391 |

## 2.6 Learning parameters using batch gradient descent

You will now find the optimal parameters of a linear regression model by using batch gradient descent. Recall batch refers to running all the examples in one iteration.

- You don't need to implement anything for this part. Simply run the cells below.
- A good way to verify that gradient descent is working correctly is to look at the value of $J(w, b)$ and check that it is decreasing with each step.
- Assuming you have implemented the gradient and computed the cost correctly and you have an appropriate value for the learning rate alpha, $J(w, b)$ should never increase and should converge to a steady value by the end of the algorithm.

```
In  [61]: /def gradient_descent(x, y, w_in, b_in, cost_function, gradient_function, alpha, nu
              """
              Performs batch gradient descent to learn theta. Updates theta by taking
              num_iters gradient steps with learning rate alpha

              Args:
                x :    (ndarray): Shape (m,)
                y :    (ndarray): Shape (m,)
                w_in, b_in : (scalar) Initial values of parameters of the model
                cost_function: function to compute cost
                gradient_function: function to compute the gradient
                alpha : (float) Learning rate
                num_iters : (int) number of iterations to run gradient descent
              Returns
                w : (ndarray): Shape (1,) Updated values of parameters of the model after
                    running gradient descent
                b : (scalar)                Updated value of parameter of the model after
                    running gradient descent
              """

              # number of training examples
              m = len(x)

              # An array to store cost J and w's at each iteration — primarily for graphing
              J_history = []
              w_history = []
              w = copy.deepcopy(w_in)  #avoid modifying global w within function
              b = b_in

              for i in range(num_iters):

                  # Calculate the gradient and update the parameters
                  dj_dw, dj_db = gradient_function(x, y, w, b )

                  # Update Parameters using w, b, alpha and gradient
                  w = w - alpha * dj_dw
                  b = b - alpha * dj_db

                  # Save cost J at each iteration
                  if i<100000:      # prevent resource exhaustion
                      cost =  cost_function(x, y, w, b)
                      J_history.append(cost)

                  # Print cost every at intervals 10 times or as many iterations if < 10
                  if i% math.ceil(num_iters/10) == 0:
                      w_history.append(w)
                      print(f"Iteration {i:4}: Cost {float(J_history[-1]):8.2f}   ")
                  /
              return w, b, J_history, w_history #return w and J,w history for graphing
```

Now let's run the gradient descent algorithm above to learn the parameters for our dataset.

In [62]:
```
# initialize fitting parameters. Recall that the shape of w is (n,)
initial_w = 0.
initial_b = 0.

# some gradient descent settings
iterations = 1500
alpha = 0.01

w,b,_,_ = gradient_descent(x_train ,y_train, initial_w, initial_b,
                     compute_cost, compute_gradient, alpha, iterations)
print("w,b found by gradient descent:", w, b)
```

```
Iteration    0: Cost     6.74
Iteration  150: Cost     5.31
Iteration  300: Cost     4.96
Iteration  450: Cost     4.76
Iteration  600: Cost     4.64
Iteration  750: Cost     4.57
Iteration  900: Cost     4.53
Iteration 1050: Cost     4.51
Iteration 1200: Cost     4.50
Iteration 1350: Cost     4.49
w,b found by gradient descent: 1.166362350335582 -3.63029143940436
```

**Expected Output**:

| | |
|---|---|
| **w, b found by gradient descent** | 1.16636235 -3.63029143940436 |

We will now use the final parameters from gradient descent to plot the linear fit.

Recall that we can get the prediction for a single example $f(x^{(i)}) = wx^{(i)} + b$.

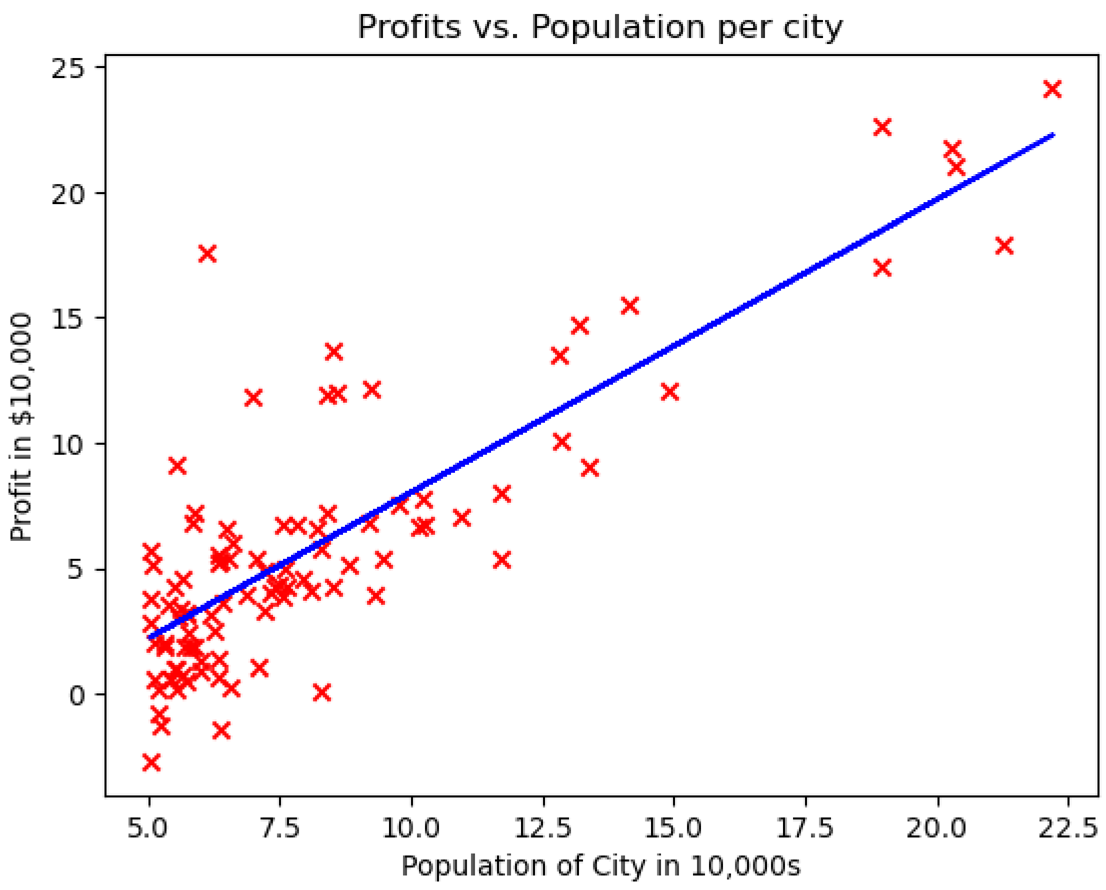
To calculate the predictions on the entire dataset, we can loop through all the training examples and calculate the prediction for each example. This is shown in the code block below.

In [63]:
```
m = x_train.shape[0]
predicted = np.zeros(m)

for i in range(m):
    predicted[i] = w * x_train[i] + b
```

We will now plot the predicted values to see the linear fit.

In [64]:
```
# Plot the linear fit
plt.plot(x_train, predicted, c = "b")

# Create a scatter plot of the data.
plt.scatter(x_train, y_train, marker='x', c='r')

# Set the title
plt.title("Profits vs. Population per city")
# Set the y-axis label
plt.ylabel('Profit in $10,000')
# Set the x-axis label
plt.xlabel('Population of City in 10,000s')
```

Out[64]:
```
Text(0.5, 0, 'Population of City in 10,000s')
```



Your final values of $w, b$ can also be used to make predictions on profits. Let's predict what the profit would be in areas of 35,000 and 70,000 people.

- The model takes in population of a city in 10,000s as input.
- Therefore, 35,000 people can be translated into an input to the model as `np.array([3.5])`
- Similarly, 70,000 people can be translated into an input to the model as `np.array([7.])`

In [65]:
```
predict1 = 3.5 * w + b
print('For population = 35,000, we predict a profit of $%.2f' % (predict1*10000))

predict2 = 7.0 * w + b
print('For population = 70,000, we predict a profit of $%.2f' % (predict2*10000))
```

```
For population = 35,000, we predict a profit of $4519.77
For population = 70,000, we predict a profit of $45342.45
```

**Expected Output**:

|  |  |
|---|---|
| **For population = 35,000, we predict a profit of** | $4519.77 |
| **For population = 70,000, we predict a profit of** | $45342.45 |

# 实验二 逻辑回归

## 一、实验目的

1. 掌握本地Jupyter notebook编译环境；
2. 熟悉Python编程集成环境Pychram、vs code；
3. 掌握基于numpy库的逻辑回归模型搭建。 本实验代码于吴恩达老师Coursera中的机器学习专项课程。

## 二、实验内容

1. 安装并配置本地Jupyter notebook编译环境；
2. 基于numpy库实现逻辑回归的代价函数、梯度下降函数等。

## 三、实验结果

1. 完成本实验中的Exercise 1-6, 直到显示All tests passed!;

## 四、实验心得(500字以内)

实验心得：通过本次实验，我利用Numpy提供的库函数，编写了sigmoid()函数,以及逻辑回归的损失函数和梯度函数。类比线性回归来说逻辑回归的策略和优化过程有了很大的不同。其激活函数不再是线性的，取而代之的是sigmoid函数，其意义是为了避免在处理分类问题上，在间断点不可导的问题。逻辑回归的代价函数也设计的很巧妙，它通过模型的预测值和实际值的乘积作为代价，从而取代线性回归中预测值和实际值之间的距离作为代价。根据代价函数，构造梯度函数，进行梯度下降，从而得到一条线性的分类模型。如果对线性逻辑回归的大家函数进行正则化改写，消解模型的复杂度，则可以实现曲线型的分类模型。

# Logistic Regression

In this exercise, you will implement logistic regression and apply it to two different datasets.

# Outline

## 1 - Packages

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

- numpy (www.numpy.org) is the fundamental package for scientific computing with Python.
- matplotlib (http://matplotlib.org) is a famous library to plot graphs in Python.
- `utils.py` contains helper functions for this assignment. You do not need to modify code in this file.

In [352]:
```
import numpy as np
import matplotlib.pyplot as plt
from utils import *
import copy
import math

%matplotlib inline
```

## 2 - Logistic Regression

In this part of the exercise, you will build a logistic regression model to predict whether a student gets admitted into a university.

### 2.1 Problem Statement

Suppose that you are the administrator of a university department and you want to determine each applicant's chance of admission based on their results on two exams.

- You have historical data from previous applicants that you can use as a training set for logistic regression.
- For each training example, you have the applicant's scores on two exams and the admissions decision.
- Your task is to build a classification model that estimates an applicant's probability of admission based on the scores from those two exams.

### 2.2 Loading and visualizing the data

You will start by loading the dataset for this task.

- The `load_dataset()` function shown below loads the data into variables `X_train` and `y_train`
  - `X_train` contains exam scores on two exams for a student

  - `y_train` is the admission decision
    - `y_train = 1` if the student was admitted
    - `y_train = 0` if the student was not admitted
  - Both `X_train` and `y_train` are numpy arrays.

In [353]:
```
# load dataset
X_train, y_train = load_data("data/ex2data1.txt")
```

#### View the variables

Let's get more familiar with your dataset.

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

The code below prints the first five values of `X_train` and the type of the variable.

In [354]:
```
print("First five elements in X_train are:\n", X_train[:5])
print("Type of X_train:",type(X_train))
```

```
First five elements in X_train are:
 [[34.62365962 78.02469282]
 [30.28671077 43.89499752]
 [35.84740877 72.90219803]
 [60.18259939 86.3085521 ]
 [79.03273605 75.34437644]]
Type of X_train: <class 'numpy.ndarray'>
```

Now print the first five values of `y_train`

In [355]:
```
print("First five elements in y_train are:\n", y_train[:5])
print("Type of y_train:",type(y_train))
```

```
First five elements in y_train are:
 [0. 0. 0. 1. 1.]
Type of y_train: <class 'numpy.ndarray'>
```

#### Check the dimensions of your variables

Another useful way to get familiar with your data is to view its dimensions. Let's print the shape of `X_train` and `y_train` and see how many training examples we have in our dataset.

In [356]:
```
print ('The shape of X_train is: ' + str(X_train.shape))
print ('The shape of y_train is: ' + str(y_train.shape))
print ('We have m = %d training examples' % (len(y_train)))
```

```
The shape of X_train is: (100, 2)
The shape of y_train is: (100,)
We have m = 100 training examples
```

#### Visualize your data

Before starting to implement any learning algorithm, it is always good to visualize the data if possible.

- The code below displays the data on a 2D plot (as shown below), where the axes are the two exam scores, and the positive and negative examples are shown with different markers.
- We use a helper function in the `utils.py` file to generate this plot.

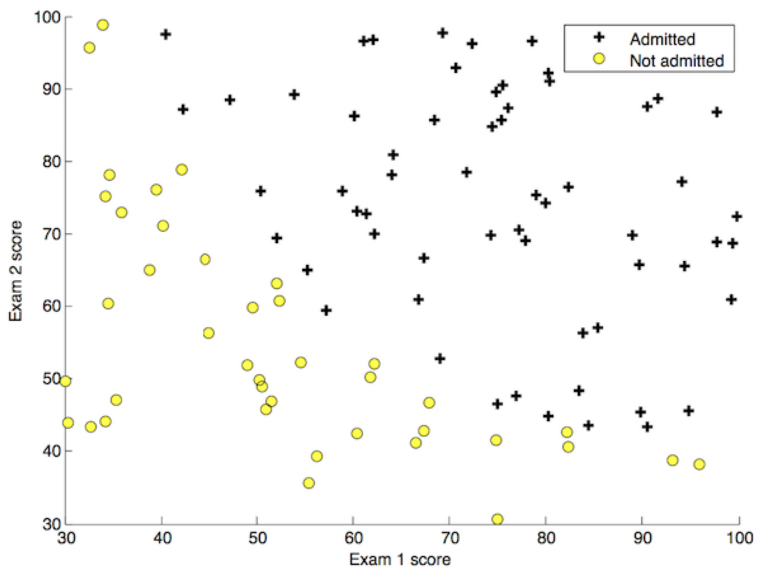
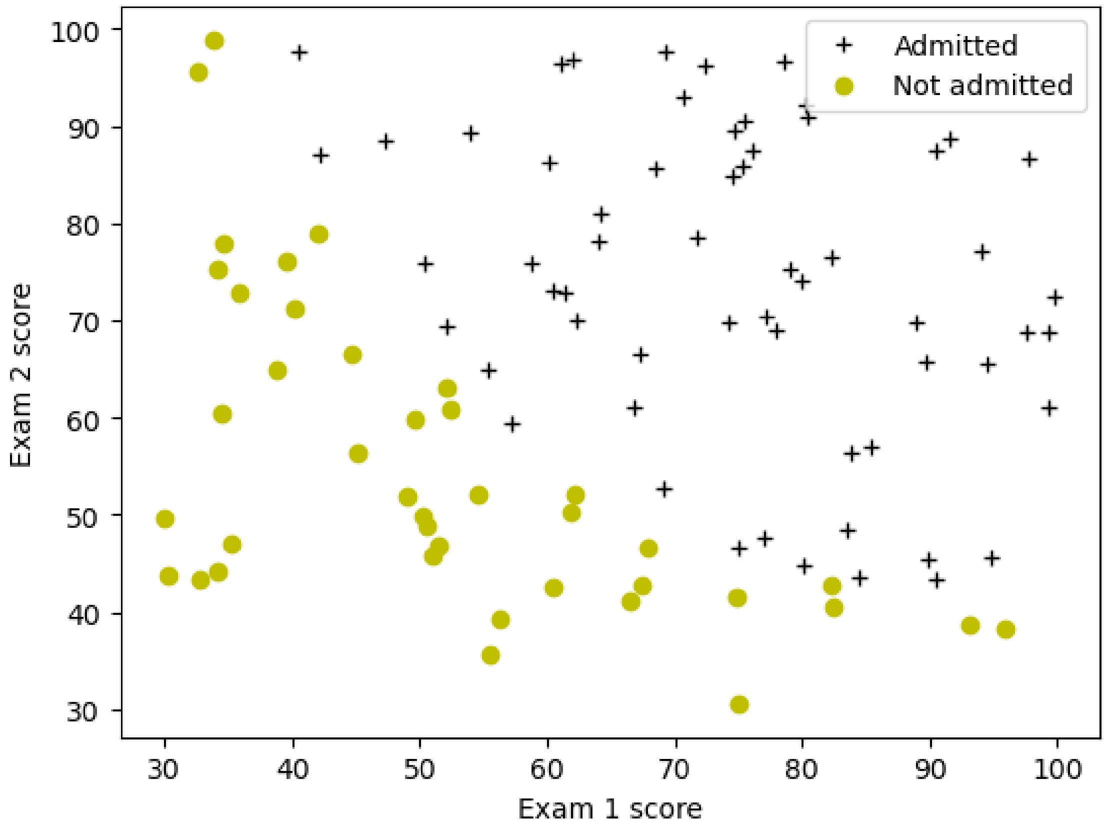


Figure 1: Scatter plot of training data

In [357]:
```
# Plot examples
plot_data(X_train, y_train[:], pos_label="Admitted", neg_label="Not admitted")

# Set the y-axis label
plt.ylabel('Exam 2 score')
# Set the x-axis label
plt.xlabel('Exam 1 score')
plt.legend(loc="upper right")
plt.show()
```



Your goal is to build a logistic regression model to fit this data.

- With this model, you can then predict if a new student will be admitted based on their scores on the two exams.

## 2.3 Sigmoid function

Recall that for logistic regression, the model is represented as

$$f_{\mathbf{w},b}(x) = g(\mathbf{w} \cdot \mathbf{x} + b)$$

where function $g$ is the sigmoid function. The sigmoid function is defined as:

$$g(z) = \frac{1}{1 + e^{-z}}$$

Let's implement the sigmoid function first, so it can be used by the rest of this assignment.

## Exercise 1

Please complete the `sigmoid` function to calculate

$$g(z) = \frac{1}{1+e^{-z}}$$

Note that

- `z` is not always a single number, but can also be an array of numbers.
- If the input is an array of numbers, we'd like to apply the sigmoid function to each value

In [358]:
```python
# UNQ_C1
# 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

    """

    ### START CODE HERE ###
    g=1/(1+np.exp(-z))
    ### END SOLUTION ###

    return g
```

When you are finished, try testing a few values by calling `sigmoid(x)` in the cell below.

- For large positive values of x, the sigmoid should be close to 1, while for large negative values, the sigmoid should be close to 0.
- Evaluating `sigmoid(0)` should give you exactly 0.5.

In [359]:
```python
print ("sigmoid(0) = " + str(sigmoid(0)))
```

```
sigmoid(0) = 0.5
```

**Expected Output**:

| | |
|---|---|
| **sigmoid(0)** | 0.5 |

- As mentioned before, your code should also work with vectors and matrices. For a matrix, your function should perform the sigmoid function on every element.

```
In [360]: print ("sigmoid([ -1, 0, 1, 2]) = " + str(sigmoid(np.array([-1, 0, 1, 2]))))

          # UNIT TESTS
          from public_tests import *
          sigmoid_test(sigmoid)
```

```
sigmoid([ -1, 0, 1, 2]) = [0.26894142 0.5        0.73105858 0.88079708]
All tests passed!
```

**Expected Output**:

|  |  |
|---|---|
| **sigmoid([-1, 0, 1, 2])** | [0.26894142 0.5 0.73105858 0.88079708] |

## 2.4 Cost function for logistic regression

In this section, you will implement the cost function for logistic regression.

## Exercise 2

Please complete the `compute_cost` function using the equations below.

Recall that for logistic regression, the cost function is of the form

$$J(\mathbf{w},b) = \frac{1}{m}\sum_{i=0}^{m-1} \left[ loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) \right] \tag{1}$$

where

- m is the number of training examples in the dataset
- $loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)})$ is the cost for a single data point, which is -

  $$loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) = (-y^{(i)} \log\left(f_{\mathbf{w},b}\left( \mathbf{x}^{(i)} \right) \right) - \left( 1 - y^{(i)}\right) \log \left( 1 - f_{\mathbf{w},b}\left( \mathbf{x}^{(i)} \right) \right)$$
- $f_{\mathbf{w},b}(\mathbf{x}^{(i)})$ is the model's prediction, while $y^{(i)}$, which is the actual label
- $f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = g(\mathbf{w} \cdot \mathbf{x^{(i)}} + b)$ where function $g$ is the sigmoid function.
  - It might be helpful to first calculate an intermediate variable $z_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w} \cdot \mathbf{x^{(i)}} + b = w_0x^{(i)}_0 + ... + w_{n-1}x^{(i)}_{n-1} + b$ where $n$ is the number of features, before calculating $f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = g(z_{\mathbf{w},b}(\mathbf{x}^{(i)}))$

Note:

- As you are doing this, remember that the variables `X_train` and `y_train` are not scalar values but matrices of shape ($m, n$) and ($m$,1) respectively, where $n$ is the number of features and $m$ is the number of training examples.
- You can use the sigmoid function that you implemented above for this part.

In [361]:
```python
# UNQ_C2
# GRADED FUNCTION: compute_cost
def compute_cost(X, y, w, b, lambda_= 1):
    """
    Computes the cost over all examples
    Args:
      X : (ndarray Shape (m,n)) data, m examples by n features
      y : (array_like Shape (m,)) target value
      w : (array_like Shape (n,)) Values of parameters of the model
      b : scalar Values of bias parameter of the model
      lambda_: unused placeholder
    Returns:
      total_cost: (scalar)         cost
    """



    ### START CODE HERE ###
    m = X.shape[0]
    cost = 0.0
    z=X@w+b
    f_wb=sigmoid(z)
    f_wb_a=np.log(f_wb)
    f_wb_b=np.log(1-f_wb)
    loss=f_wb_a*(-y)-f_wb_b*(1-y)
    cost=np.sum(loss)/m
    ### END CODE HERE ###

    return cost
```

Run the cells below to check your implementation of the `compute_cost` function with two different initializations of the parameters $w$

In [362]:
```python
m, n = X_train.shape

# Compute and display cost with w initialized to zeroes
initial_w = np.zeros(n)
initial_b = 0.
cost = compute_cost(X_train, y_train, initial_w, initial_b)
print('Cost at initial w (zeros): {:.3f}'.format(cost))
```

```
Cost at initial w (zeros): 0.693
```

**Expected Output**:

| | |
|---|---|
| **Cost at initial w (zeros)** | 0.693 |

In [363]:

```
# Compute and display cost with non-zero w
test_w = np.array([0.2, 0.2])
test_b = -24.
cost = compute_cost(X_train, y_train, test_w, test_b)

print('Cost at test w,b: {:.3f}'.format(cost))


# UNIT TESTS
compute_cost_test(compute_cost)
```

```
Cost at test w,b: 0.218
All tests passed!
```

In [ ]:

```

```

**Expected Output**:

| | |
|---|---|
| **Cost at test w,b** | 0.218 |

## 2.5 Gradient for logistic regression

In this section, you will implement the gradient for logistic regression.

Recall that the gradient descent algorithm is:

$$\begin{align*}&\text{repeat until convergence:} \; \lbrace \\ &  b := b -  \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \\ &  w_j := w_j -  \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \tag{1}  \; & \text{for j := 0..n-1}\\ & \rbrace\end{align*}$$

where, parameters $b$, $w_j$ are all updated simultaniously

## Exercise 3

Please complete the `compute_gradient` function to compute $\frac{\partial J(\mathbf{w},b)}{\partial w}$, $\frac{\partial J(\mathbf{w},b)}{\partial b}$ from equations (2) and (3) below.

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}) \tag{2}$$

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)})x_{j}^{(i)} \tag{3}$$

- m is the number of training examples in the dataset
- $f_{\mathbf{w},b}(x^{(i)})$ is the model's prediction, while $y^{(i)}$ is the actual label

- **Note**: While this gradient looks identical to the linear regression gradient, the formula is actually different because linear and logistic regression have different definitions of $f_{\mathbf{w},b}(x)$.

As before, you can use the sigmoid function that you implemented above and if you get stuck.

In [364]:
```
# UNQ_C3
# GRADED FUNCTION: compute_gradient
def compute_gradient(X, y, w, b,lambda_=None):
    """
    Computes the gradient for logistic regression

    Args:
      X : (ndarray Shape (m,n)) variable such as house size
      y : (array_like Shape (m,1)) actual value
      w : (array_like Shape (n,1)) values of parameters of the model
      b : (scalar)                 value of parameter of the model
      lambda_: unused placeholder.
    Returns
      dj_dw: (array_like Shape (n,1)) The gradient of the cost w.r.t. the parameter
      dj_db: (scalar)                The gradient of the cost w.r.t. the parameter
    """
    m, n = X.shape
    dj_dw = np.zeros(w.shape)
    dj_db = 0.

    ### START CODE HERE ###
    for i in range(m):
        z_wb = 0
        for j in range(n):
            z_wb += w[j]*X[i,j]
        z_wb += b
        f_wb = sigmoid(z_wb)

        dj_db_i = f_wb-y[i]
        dj_db += dj_db_i

        for j in range(n):
            dj_dw[j] += dj_db_i*X[i,j]

    dj_dw = dj_dw/m
    dj_db = dj_db/m
    ### END CODE HERE ###


    return dj_db, dj_dw
```

Run the cells below to check your implementation of the `compute_gradient` function with two different initializations of the parameters $w$

In [365]:
```
# Compute and display gradient with w initialized to zeroes
initial_w = np.zeros(n)
initial_b = 0.

dj_db, dj_dw = compute_gradient(X_train, y_train, initial_w, initial_b)
print(f'dj_db at initial w (zeros):{dj_db}' )
print(f'dj_dw at initial w (zeros):{dj_dw.tolist()}' )
```

```
dj_db at initial w (zeros):-0.1
dj_dw at initial w (zeros):[-12.00921658929115, -11.262842205513591]
```

**Expected Output**:

| | |
|---|---|
| **dj_db at initial w (zeros)** | -0.1 |
| **ddj_dw at initial w (zeros):** | [-12.00921658929115, -11.262842205513591] |

In [366]:
```
# Compute and display cost and gradient with non-zero w
test_w = np.array([ 0.2, -0.5])
test_b = -24
dj_db, dj_dw  = compute_gradient(X_train, y_train, test_w, test_b)

print('dj_db at test_w:', dj_db)
print('dj_dw at test_w:', dj_dw.tolist())

# UNIT TESTS
compute_gradient_test(compute_gradient)
```

```
dj_db at test_w: -0.5999999999991071
dj_dw at test_w: [-44.831353617873795, -44.37384124953978]
All tests passed!
```

**Expected Output**:

| | |
|---|---|
| **dj_db at initial w (zeros)** | -0.5999999999991071 |
| **ddj_dw at initial w (zeros):** | [-44.8313536178737957, -44.37384124953978] |

## 2.6 Learning parameters using gradient descent

Similar to the previous assignment, you will now find the optimal parameters of a logistic regression model by using gradient descent.

- You don't need to implement anything for this part. Simply run the cells below.
- A good way to verify that gradient descent is working correctly is to look at the value of $J(\mathbf{w}, b)$ and check that it is decreasing with each step.
- Assuming you have implemented the gradient and computed the cost correctly, your value of $J(\mathbf{w}, b)$ should never increase, and should converge to a steady value by the end of the algorithm.

```
In [367]: def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function, alpha, nur
              """
              Performs batch gradient descent to learn theta. Updates theta by taking
              num_iters gradient steps with learning rate alpha

              Args:
                X :    (array_like Shape (m, n)
                y :    (array_like Shape (m,))
                w_in : (array_like Shape (n,))  Initial values of parameters of the model
                b_in : (scalar)                 Initial value of parameter of the model
                cost_function:                  function to compute cost
                alpha : (float)                 Learning rate
                num_iters : (int)               number of iterations to run gradient descent
                lambda_ (scalar, float)         regularization constant

              Returns:
                w : (array_like Shape (n,)) Updated values of parameters of the model after
                    running gradient descent
                b : (scalar)                Updated value of parameter of the model after
                    running gradient descent
              """

              # number of training examples
              m = len(X)

              # An array to store cost J and w's at each iteration primarily for graphing lat
              J_history = []
              w_history = []

              for i in range(num_iters):

                  # Calculate the gradient and update the parameters
                  dj_db, dj_dw = gradient_function(X, y, w_in, b_in, lambda_)

                  # Update Parameters using w, b, alpha and gradient
                  w_in = w_in - alpha * dj_dw
                  b_in = b_in - alpha * dj_db

                  # Save cost J at each iteration
                  if i<100000:      # prevent resource exhaustion
                      cost =  cost_function(X, y, w_in, b_in, lambda_)
                      J_history.append(cost)

                  # Print cost every at intervals 10 times or as many iterations if < 10
                  if i% math.ceil(num_iters/10) == 0 or i == (num_iters-1):
                      w_history.append(w_in)
                      print(f"Iteration {i:4}: Cost {float(J_history[-1]):8.2f}   ")

              return w_in, b_in, J_history, w_history #return w and J,w history for graphing
```

Now let's run the gradient descent algorithm above to learn the parameters for our dataset.

**Note**

The code block below takes a couple of minutes to run, especially with a non-vectorized version. You can reduce the `iterations` to test your implementation and iterate faster. If you have time, try running 100,000 iterations for better results.

In [368]:
```
np.random.seed(1)
intial_w = 0.01 * (np.random.rand(2).reshape(-1,1) - 0.5)
initial_b = -8


# Some gradient descent settings
iterations = 10000
alpha = 0.001

w,b, J_history,_ = gradient_descent(X_train ,y_train, initial_w, initial_b, compute
```

```
Iteration    0: Cost     1.01
Iteration 1000: Cost     0.31
Iteration 2000: Cost     0.30
Iteration 3000: Cost     0.30
Iteration 4000: Cost     0.30
Iteration 5000: Cost     0.30
Iteration 6000: Cost     0.30
Iteration 7000: Cost     0.30
Iteration 8000: Cost     0.30
Iteration 9000: Cost     0.30
Iteration 9999: Cost     0.30
```

**Expected Output: Cost 0.30, (Click to see details):**

## 2.7 Plotting the decision boundary

We will now use the final parameters from gradient descent to plot the linear fit. If you implemented the previous parts correctly, you should see the following plot:

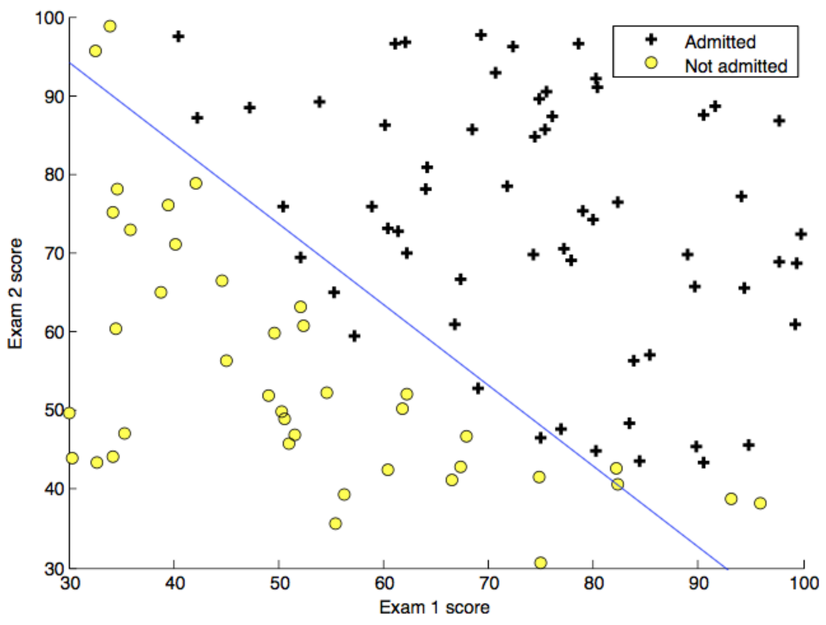
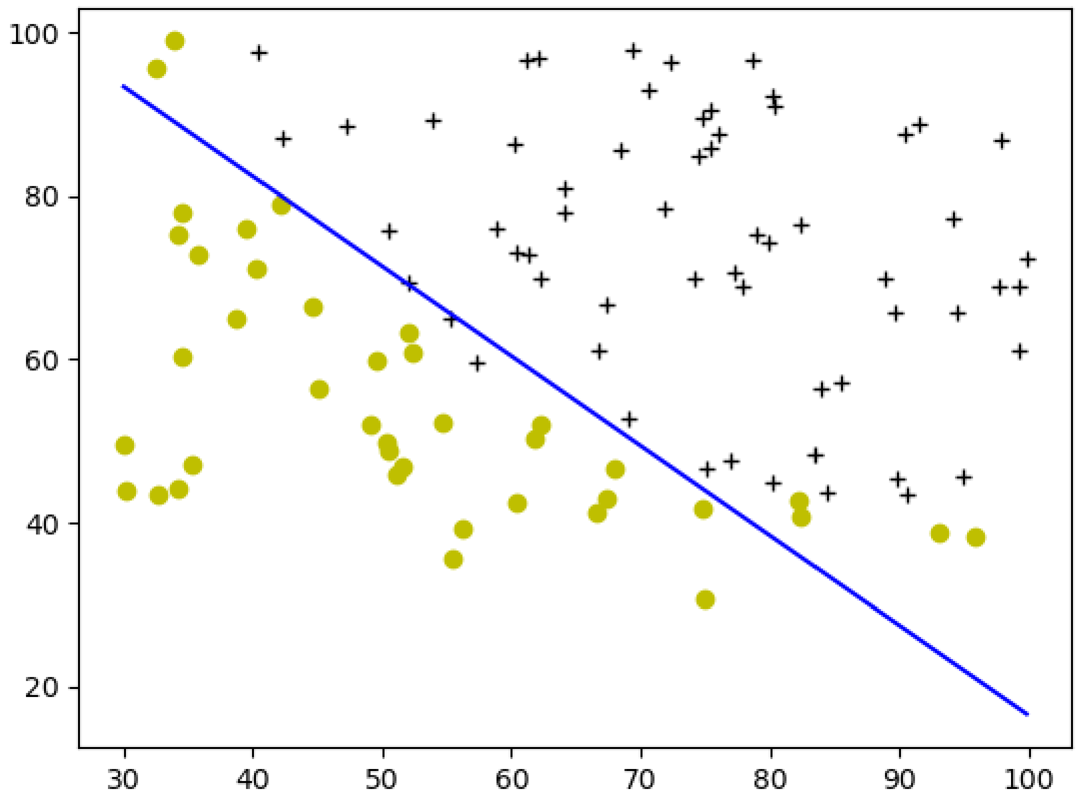


Figure 2: Training data with decision boundary

We will use a helper function in the `utils.py` file to create this plot.

In [369]:
```
plot_decision_boundary(w, b, X_train, y_train)
```



## 2.8 Evaluating logistic regression

We can evaluate the quality of the parameters we have found by seeing how well the learned model predicts on our training set.

You will implement the `predict` function below to do this.

## Exercise 4

Please complete the `predict` function to produce `1` or `0` predictions given a dataset and a learned parameter vector $w$ and $b$.

- First you need to compute the prediction from the model $f(x^{(i)}) = g(w \cdot x^{(i)})$ for every example
  - You've implemented this before in the parts above
- We interpret the output of the model ($f(x^{(i)})$) as the probability that $y^{(i)} = 1$ given $x^{(i)}$ and parameterized by $w$.
- Therefore, to get a final prediction ($y^{(i)} = 0$ or $y^{(i)} = 1$) from the logistic regression model, you can use the following heuristic -

  if $f(x^{(i)}) >= 0.5$, predict $y^{(i)} = 1$

  if $f(x^{(i)}) < 0.5$, predict $y^{(i)} = 0$

In [370]:

```
# UNQ_C4
# GRADED FUNCTION: predict

def predict(X, w, b):
    """
    Predict whether the label is 0 or 1 using learned logistic
    regression parameters w

    Args:
    X : (ndarray Shape (m, n))
    w : (array_like Shape (n,))      Parameters of the model
    b : (scalar, float)              Parameter of the model

    Returns:
    p: (ndarray (m,1))
        The predictions for X using a threshold at 0.5
    """
    # number of training examples
    m, n = X.shape
    p = np.zeros(m)

    ### START CODE HERE ###
    # Loop over each example
    for i in range(m):
        z_wb = 0
        # Loop over each feature
        for j in range(n):
            # Add the corresponding term to z_wb
            z_wb += w[j]*X[i,j]

        # Add bias term
        z_wb += b

        # Calculate the prediction for this example
        f_wb = sigmoid(z_wb)

        # Apply the threshold
        p[i] = 1 if f_wb>=0.5 else 0

    ### END CODE HERE ###
    return p
```

Once you have completed the function `predict` , let's run the code below to report the training accuracy of your classifier by computing the percentage of examples it got correct.

In [371]:
```
# Test your predict code
np.random.seed(1)
tmp_w = np.random.randn(2)
tmp_b = 0.3
tmp_X = np.random.randn(4, 2) - 0.5

tmp_p = predict(tmp_X, tmp_w, tmp_b)
print(f'Output of predict: shape {tmp_p.shape}, value {tmp_p}')

# UNIT TESTS
predict_test(predict)
```

```
Output of predict: shape (4,), value [0. 1. 1. 1.]
All tests passed!
```

**Expected output**

**Output of predict: shape (4,),value [0. 1. 1. 1.]**

Now let's use this to compute the accuracy on the training set

In [372]:
```
#Compute accuracy on our training set
p = predict(X_train, w,b)
print('Train Accuracy: %f'%(np.mean(p == y_train) * 100))
```

```
Train Accuracy: 92.000000
```

**Train Accuracy (approx):** 92.00

## 3 - Regularized Logistic Regression

In this part of the exercise, you will implement regularized logistic regression to predict whether microchips from a fabrication plant passes quality assurance (QA). During QA, each microchip goes through various tests to ensure it is functioning correctly.

### 3.1 Problem Statement

Suppose you are the product manager of the factory and you have the test results for some microchips on two different tests.

- From these two tests, you would like to determine whether the microchips should be accepted or rejected.
- To help you make the decision, you have a dataset of test results on past microchips, from which you can build a logistic regression model.

### 3.2 Loading and visualizing the data

Similar to previous parts of this exercise, let's start by loading the dataset for this task and visualizing it.

- The `load_dataset()` function shown below loads the data into variables `X_train` and `y_train`
  - `X_train` contains the test results for the microchips from two tests
  - `y_train` contains the results of the QA
    - `y_train = 1` if the microchip was accepted
    - `y_train = 0` if the microchip was rejected
  - Both `X_train` and `y_train` are numpy arrays

In [373]:
```
# load dataset
X_train, y_train = load_data("data/ex2data2.txt")
```

#### View the variables

The code below prints the first five values of `X_train` and `y_train` and the type of the variables.

In [374]:
```
# print X_train
print("X_train:", X_train[:5])
print("Type of X_train:",type(X_train))

# print y_train
print("y_train:", y_train[:5])
print("Type of y_train:",type(y_train))
```

```
X_train: [[ 0.051267  0.69956 ]
 [-0.092742  0.68494 ]
 [-0.21371   0.69225 ]
 [-0.375     0.50219 ]
 [-0.51325   0.46564 ]]
Type of X_train: <class 'numpy.ndarray'>
y_train: [1. 1. 1. 1. 1.]
Type of y_train: <class 'numpy.ndarray'>
```

#### Check the dimensions of your variables

Another useful way to get familiar with your data is to view its dimensions. Let's print the shape of `X_train` and `y_train` and see how many training examples we have in our dataset.

In [375]:
```
print ('The shape of X_train is: ' + str(X_train.shape))
print ('The shape of y_train is: ' + str(y_train.shape))
print ('We have m = %d training examples' % (len(y_train)))
```

```
The shape of X_train is: (118, 2)
The shape of y_train is: (118,)
We have m = 118 training examples
```

#### Visualize your data

The helper function `plot_data` (from `utils.py` ) is used to generate a figure like Figure 3, where the axes are the two test scores, and the positive (y = 1, accepted) and negative (y = 0, rejected) examples are shown with different markers.

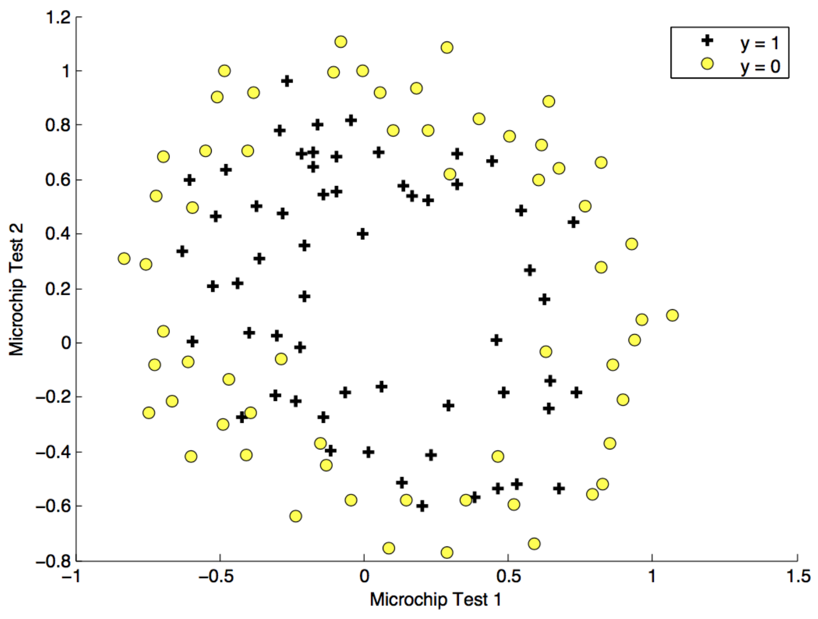


Figure 3: Plot of training data

```
In  [376]: # Plot examples
           plot_data(X_train, y_train[:], pos_label="Accepted", neg_label="Rejected")

           # Set the y-axis label
           plt.ylabel('Microchip Test 2')
           # Set the x-axis label
           plt.xlabel('Microchip Test 1')
           plt.legend(loc="upper right")
           plt.show()
```

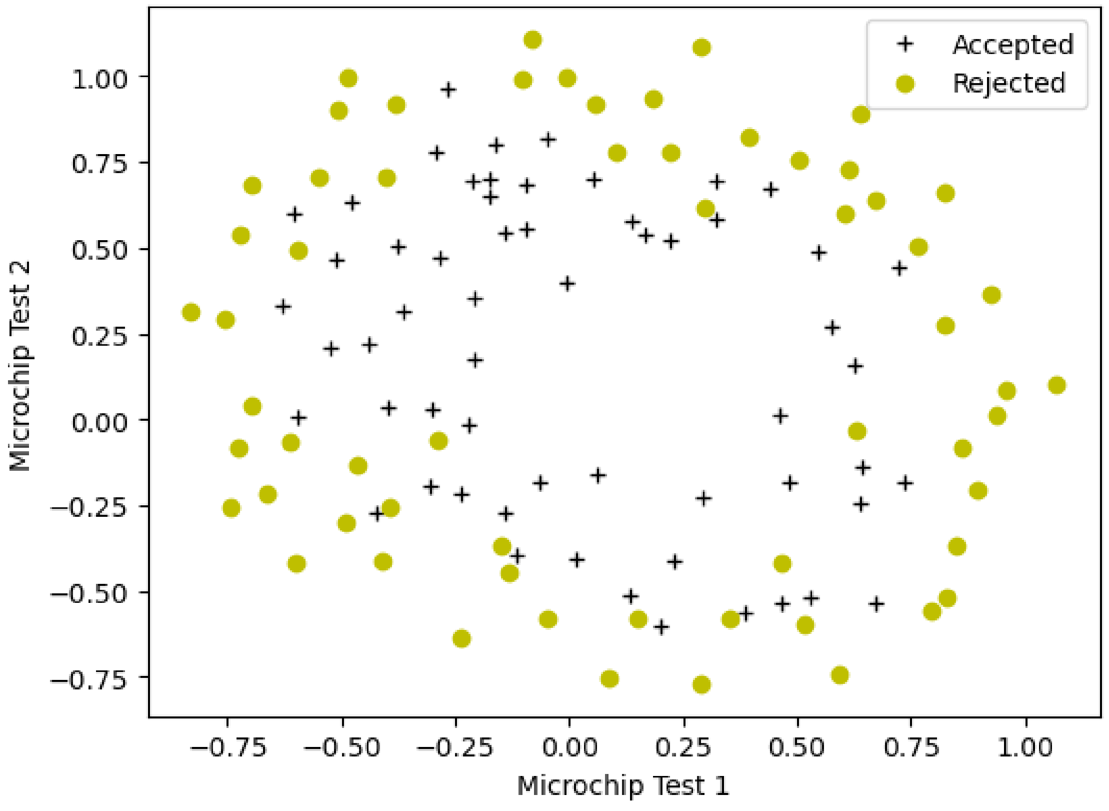


Figure 3 shows that our dataset cannot be separated into positive and negative examples by a straight-line through the plot. Therefore, a straight forward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

## 3.3 Feature mapping

One way to fit the data better is to create more features from each data point. In the provided function `map_feature`, we will map the features into all polynomial terms of $x_1$ and $x_2$ up to the sixth power.

$$\text{map\_feature}(x) = \left[\begin{array}{c} x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1 x_2^5 \\ x_2^6 \end{array}\right]$$

As a result of this mapping, our vector of two features (the scores on two QA tests) has been transformed into a 27-dimensional vector.

- A logistic regression classifier trained on this higher-dimension feature vector will have a more complex decision boundary and will be nonlinear when drawn in our 2-dimensional plot.
- We have provided the `map_feature` function for you in utils.py.

In [377]:
```
print("Original shape of data:", X_train.shape)

mapped_X =  map_feature(X_train[:, 0], X_train[:, 1])
print("Shape after feature mapping:", mapped_X.shape)
```

```
Original shape of data: (118, 2)
Shape after feature mapping: (118, 27)
```

Let's also print the first elements of `X_train` and `mapped_X` to see the tranformation.

```
In  [378]: print("X_train[0]:", X_train[0])
           print("mapped X_train[0]:", mapped_X[0])
```

```
X_train[0]: [0.051267 0.69956 ]
mapped X_train[0]: [5.12670000e-02 6.99560000e-01 2.62830529e-03 3.58643425e-02
 4.89384194e-01 1.34745327e-04 1.83865725e-03 2.50892595e-02
 3.42353606e-01 6.90798869e-06 9.42624411e-05 1.28625106e-03
 1.75514423e-02 2.39496889e-01 3.54151856e-07 4.83255257e-06
 6.59422333e-05 8.99809795e-04 1.22782870e-02 1.67542444e-01
 1.81563032e-08 2.47750473e-07 3.38066048e-06 4.61305487e-05
 6.29470940e-04 8.58939846e-03 1.17205992e-01]
```

While the feature mapping allows us to build a more expressive classifier, it is also more susceptible to overfitting. In the next parts of the exercise, you will implement regularized logistic regression to fit the data and also see for yourself how regularization can help combat the overfitting problem.

## 3.4 Cost function for regularized logistic regression

In this part, you will implement the cost function for regularized logistic regression.

Recall that for regularized logistic regression, the cost function is of the form

$$J(\mathbf{w},b) = \frac{1}{m} \sum_{i=0}^{m-1} \left[ -y^{(i)} \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) \right] + \frac{\lambda}{2m} \sum_{j=0}^{n-1} w$$

Compare this to the cost function without regularization (which you implemented above), which is of the form

$$J(\mathbf{w}.b) = \frac{1}{m}\sum_{i=0}^{m-1} \left[ (-y^{(i)} \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right)\right]$$

The difference is the regularization term, which is

$$\frac{\lambda}{2m} \sum_{j=0}^{n-1} w_j^2$$

Note that the $b$ parameter is not regularized.

## Exercise 5

Please complete the `compute_cost_reg` function below to calculate the following term for each element in $w$

$$\frac{\lambda}{2m} \sum_{j=0}^{n-1} w_j^2$$

The starter code then adds this to the cost without regularization (which you computed above in `compute_cost` ) to calculate the cost with regulatization.

```
In [379]: # UNQ_C5
          def compute_cost_reg(X, y, w, b, lambda_ = 1):
              """
              Computes the cost over all examples
              Args:
                X : (array_like Shape (m,n)) data, m examples by n features
                y : (array_like Shape (m,)) target value
                w : (array_like Shape (n,)) Values of parameters of the model
                b : (array_like Shape (n,)) Values of bias parameter of the model
                lambda_ : (scalar, float)    Controls amount of regularization
              Returns:
                total_cost: (scalar)         cost
              """

              m, n = X.shape

              # Calls the compute_cost function that you implemented above
              cost_without_reg = compute_cost(X, y, w, b)

              # You need to calculate this value
              reg_cost = 0.

              ### START CODE HERE ###
              for j in range (n):
                  reg_cost+=w[j]*w[j]

              ### END CODE HERE ###

              # Add the regularization cost to get the total cost
              total_cost = cost_without_reg + (lambda_/(2 * m)) * reg_cost

              return total_cost
```

Run the cell below to check your implementation of the `compute_cost_reg` function.

```
In [380]: X_mapped = map_feature(X_train[:, 0], X_train[:, 1])
          np.random.seed(1)
          initial_w = np.random.rand(X_mapped.shape[1]) - 0.5
          initial_b = 0.5
          lambda_ = 0.5
          cost = compute_cost_reg(X_mapped, y_train, initial_w, initial_b, lambda_)

          print("Regularized cost :", cost)

          # UNIT TEST
          compute_cost_reg_test(compute_cost_reg)
```

```
Regularized cost : 0.6618252552483951
All tests passed!
```

**Expected Output**:

| | |
|---|---|
| **Regularized cost :** | 0.6618252552483948 |

## 3.5 Gradient for regularized logistic regression

In this section, you will implement the gradient for regularized logistic regression.

The gradient of the regularized cost function has two components. The first, $\frac{\partial J(\mathbf{w},b)}{\partial b}$ is a scalar, the other is a vector with the same shape as the parameters $\mathbf{w}$, where the $j^{\mathrm{th}}$ element is defined as follows:

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})$$

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \left( \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} w_j \quad \text{for } j = 0...(n-1)$$

Compare this to the gradient of the cost function without regularization (which you implemented above), which is of the form

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}) \tag{2}$$

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}) x_j^{(i)} \tag{3}$$

As you can see, $\frac{\partial J(\mathbf{w},b)}{\partial b}$ is the same, the difference is the following term in $\frac{\partial J(\mathbf{w},b)}{\partial w}$, which is

$$\frac{\lambda}{m} w_j \quad \text{for } j = 0...(n-1)$$

## Exercise 6

Please complete the `compute_gradient_reg` function below to modify the code below to calculate the following term

$$\frac{\lambda}{m} w_j \quad \text{for } j = 0...(n-1)$$

The starter code will add this term to the $\frac{\partial J(\mathbf{w},b)}{\partial w}$ returned from `compute_gradient` above to get the gradient for the regularized cost function.

In [381]:
```
# UNQ_C6
def compute_gradient_reg(X, y, w, b, lambda_ = 1):
    """
    Computes the gradient for linear regression

    Args:
      X : (ndarray Shape (m,n))   variable such as house size
      y : (ndarray Shape (m,))    actual value
      w : (ndarray Shape (n,))    values of parameters of the model
      b : (scalar)                value of parameter of the model
      lambda_ : (scalar,float)    regularization constant
    Returns
      dj_db: (scalar)             The gradient of the cost w.r.t. the parameter b.
      dj_dw: (ndarray Shape (n,)) The gradient of the cost w.r.t. the parameters w.

    """
    m, n = X.shape

    dj_db, dj_dw = compute_gradient(X, y, w, b)
    ### START CODE HERE ###
    g=0.0

    dj_dw+=(w*lambda_/m)
    ### END CODE HERE ###

    return dj_db, dj_dw
```

Run the cell below to check your implementation of the `compute_gradient_reg` function.

In [382]:
```
X_mapped = map_feature(X_train[:, 0], X_train[:, 1])
np.random.seed(1)
initial_w  = np.random.rand(X_mapped.shape[1]) - 0.5
initial_b = 0.5

lambda_ = 0.5
dj_db, dj_dw = compute_gradient_reg(X_mapped, y_train, initial_w, initial_b, lambda

print(f"dj_db: {dj_db}", )
print(f"First few elements of regularized dj_dw:\n {dj_dw[:4].tolist()}", )

# UNIT TESTS
compute_gradient_reg_test(compute_gradient_reg)
```

```
dj_db: 0.07138288792343662
First few elements of regularized dj_dw:
 [-0.010386028450548701, 0.011409852883280124, 0.0536273463274574, 0.00314027826
7313462]
All tests passed!
```

**Expected Output**:

**dj_db**:0.07138288792343656

**First few elements of regularized dj_dw:**

[[-0.010386028450548701], [0.01140985288328012], [0.0536273463274574], [0.0031402782673134621]]

## 3.6 Learning parameters using gradient descent

Similar to the previous parts, you will use your gradient descent function implemented above to learn the optimal parameters $w$,$b$.

- If you have completed the cost and gradient for regularized logistic regression correctly, you should be able to step through the next cell to learn the parameters $w$.
- After training our parameters, we will use it to plot the decision boundary.

**Note**

The code block below takes quite a while to run, especially with a non-vectorized version. You can reduce the `iterations` to test your implementation and iterate faster. If you have time, run for 100,000 iterations to see better results.

In [383]:
```
# Initialize fitting parameters
np.random.seed(1)
initial_w = np.random.rand(X_mapped.shape[1])-0.5
initial_b = 1.

# Set regularization parameter lambda_ to 1 (you can try varying this)
lambda_ = 0.01;
# Some gradient descent settings
iterations = 10000
alpha = 0.01

w,b, J_history,_ = gradient_descent(X_mapped, y_train, initial_w, initial_b,
                                    compute_cost_reg, compute_gradient_reg,
                                    alpha, iterations, lambda_)
```

```
Iteration    0: Cost     0.72
Iteration 1000: Cost     0.59
Iteration 2000: Cost     0.56
Iteration 3000: Cost     0.53
Iteration 4000: Cost     0.51
Iteration 5000: Cost     0.50
Iteration 6000: Cost     0.48
Iteration 7000: Cost     0.47
Iteration 8000: Cost     0.46
Iteration 9000: Cost     0.45
Iteration 9999: Cost     0.45
```

**Expected Output: Cost < 0.5 (Click for details)**

## 3.7 Plotting the decision boundary

To help you visualize the model learned by this classifier, we will use our `plot_decision_boundary` function which plots the (non-linear) decision boundary that separates the positive and negative examples.

- In the function, we plotted the non-linear decision boundary by computing the classifier's predictions on an evenly spaced grid and then drew a contour plot of where the predictions change from y = 0 to y = 1.

- After learning the parameters $w$,$b$, the next step is to plot a decision boundary similar to Figure 4.

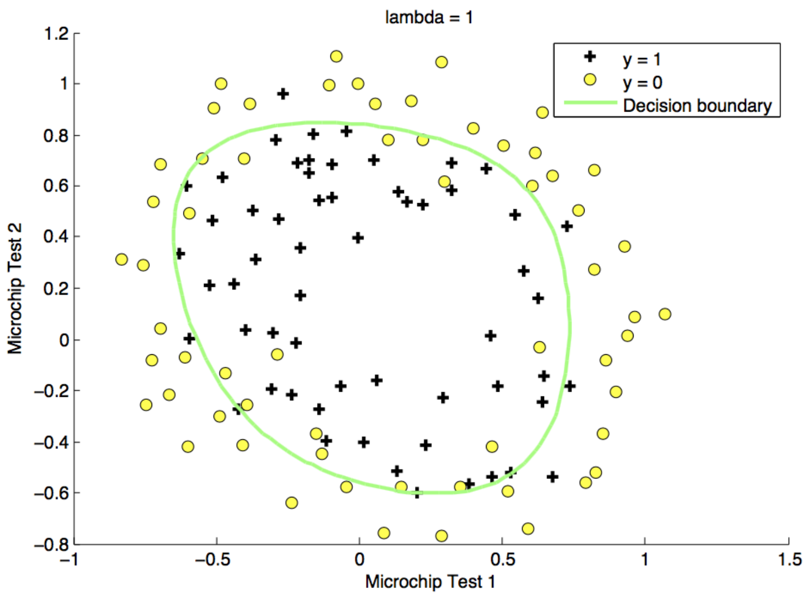
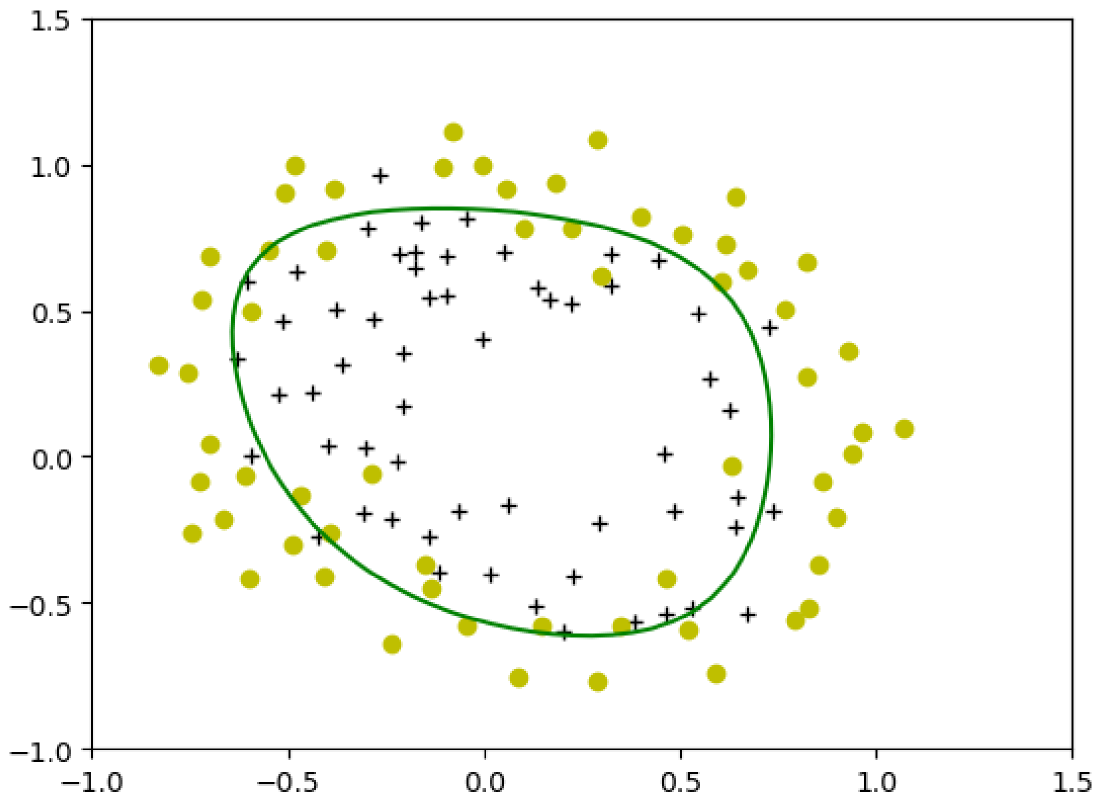


Figure 4: Training data with decision boundary ($\lambda = 1$)

In [384]:
```
plot_decision_boundary(w, b, X_mapped, y_train)
```



## 3.8 Evaluating regularized logistic regression model

You will use the `predict` function that you implemented above to calculate the accuracy of the regulaized logistic regression model on the training set

In [385]:
```
#Compute accuracy on the training set
p = predict(X_mapped, w, b)

print('Train Accuracy: %f'%(np.mean(p == y_train) * 100))
```

```
Train Accuracy: 82.203390
```

**Expected Output**:

**Train Accuracy:**~ 80%

In [ ]:
```

```

# 实验三 神经网络-二分类

## 一、实验目的

1. 掌握本地Jupyter notebook编译环境；
2. 熟悉Python编程集成环境Pychram、vs code；
3. 掌握基于numpy与Tensorflow库的神经网络模型搭建。

本实验代码于吴恩达老师Coursera中的机器学习专项课程。

## 二、实验内容

1. 安装并配置本地Jupyter notebook编译环境；
2. 基于numpy与Tensorflow库的神经网络二分类模型搭建。

## 三、实验结果

1. 完成本实验中的Exercise 1-3, 直到显示All tests passed!;

## 四、实验心得(500字以内)

实验心得：通过本次实验，我学会了如何利用 Keras Sequential 模型和具有 sigmoid 激活的 Dense Layer 来构建一个简单的神经网络。随着实验的进行，我学会了Dense函数的运行原理，并根据其原理编写了两个my_dense函数替代 Dens Layer 给的模板函数。从Dense函数参数来看，它指定了一层网络中Units的个数和本层网络的激活函数；在函数内部来说，既可以利用for循环来编写矩阵运算的过程，也可以利用Numpy提供的函数np.matmul()来直接进行矩阵间的运算。构建好每一层的神经网络后，还需要选择损失函数和梯度下降策略。在载入训练集进行20次epoch训练过后，我得到了自己的0和1分类的模型。 本节课尚存在一些疑点：为什么建立一个25×15×1的神经网络，应当如何确定自己神经网络每一层的神经元个数，希望在以后的学习中可以找到答案。

# 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

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

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

In [3]:
```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt
from autils import *
%matplotlib inline

import logging
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 (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.

$$X = \left(\begin{array}{cc} --- (x^{(1)}) --- \\ --- (x^{(2)}) --- \\ \vdots \\ --- (x^{(m)}) --- \end{array}\right)$$

- 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/ (http://yann.lecun.com/exdb/mnist/))

In [4]:
```
# load dataset
X, y = load_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 [5]: 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
  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 [6]: print ('The first element of y is: ', y[0,0])
        print ('The last element of y is: ', y[-1,0])
```

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

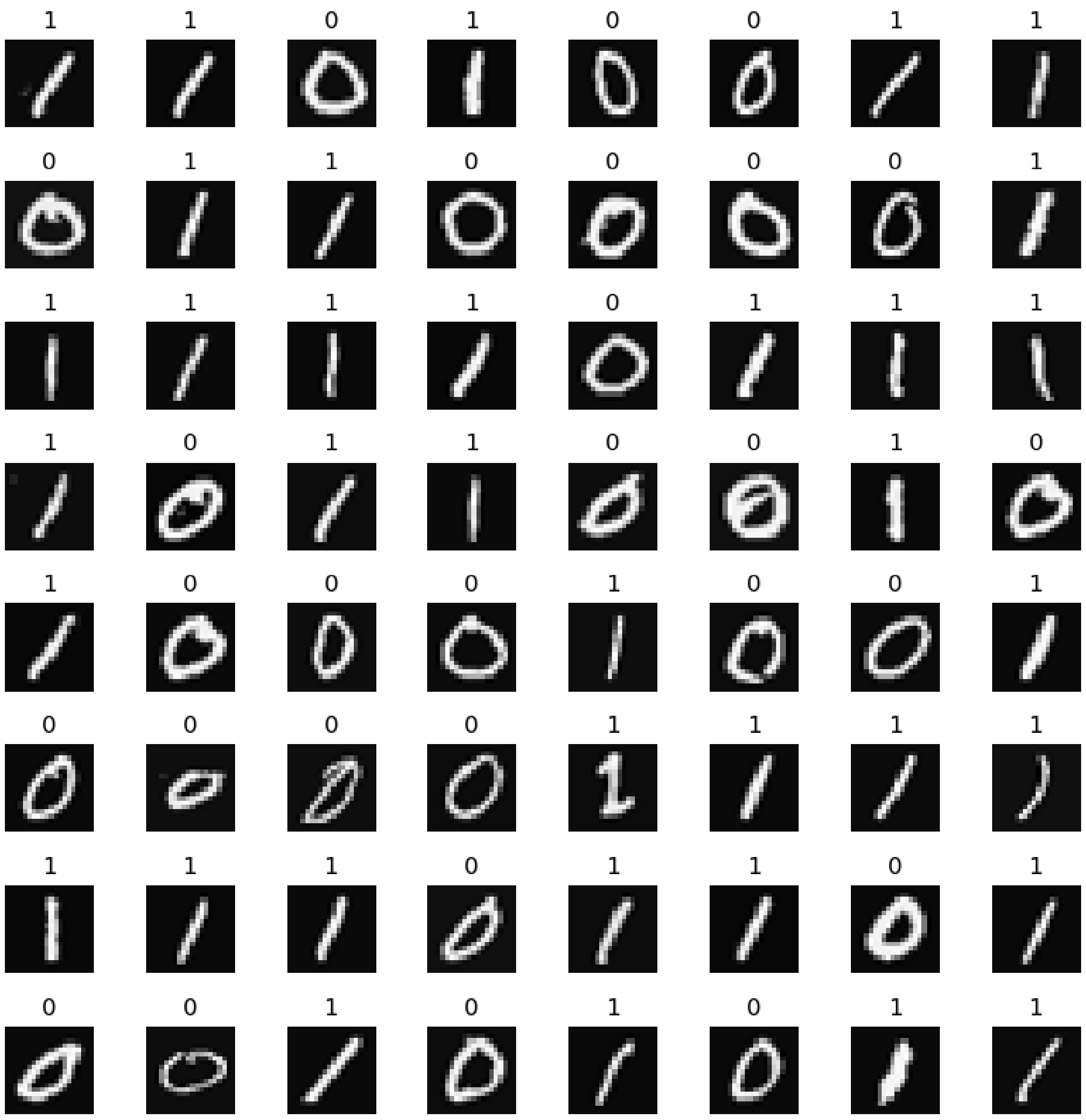
```
The shape of X is: (1000, 400)
The shape of y is: (1000, 1)
```

#### 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

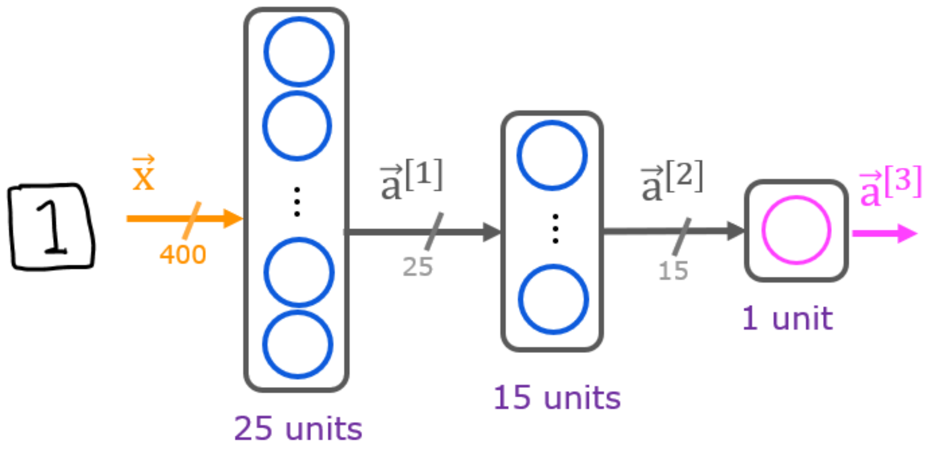
In [8]:
```python
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
# You do not need to modify anything in this cell

m, n = X.shape

fig, axes = plt.subplots(8,8, figsize=(8,8))
fig.tight_layout(pad=0.1)

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(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 $20 \times 20$, 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
      - $W$ 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 (n,1) 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` statment below.

## 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 [19]:
```
# UNQ_C1
# GRADED CELL: Sequential model

model = Sequential(
    [
        tf.keras.Input(shape=(400,)),    #specify input size
        ### START CODE HERE ###
        Dense(25,activation='sigmoid',name='layer1'),
        Dense(15,activation='sigmoid',name='layer2'),
        Dense(1,activation='sigmoid',name='layer3')
        ### END CODE HERE ###
    ], name = "my_model"
)
```

In [20]:
```
model.summary()
```

```
Model: "my_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 layer1 (Dense)              (None, 25)                10025

 layer2 (Dense)              (None, 15)                390

 layer3 (Dense)              (None, 1)                 16

=================================================================
Total params: 10431 (40.75 KB)
Trainable params: 10431 (40.75 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

### Expected Output (Click to Expand)

In [21]:
```
# UNIT 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 [22]:
```
L1_num_params = 400 * 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 =  10025 , 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 [23]:
```
[layer1, layer2, layer3] = model.layers
```

In [24]:
```
#### 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 = (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 [25]:
```
print(model.layers[2].weights)
```

```
[<tf.Variable 'layer3/kernel:0' shape=(15, 1) dtype=float32, numpy=
array([[ 0.41469365],
       [ 0.17289668],
       [ 0.6021336 ],
       [ 0.07787865],
       [-0.10991949],
       [-0.37612113],
       [ 0.08926427],
       [ 0.06583095],
       [ 0.32765186],
       [-0.45328483],
       [-0.512994  ],
       [ 0.09385496],
       [ 0.31334978],
       [-0.43881935],
       [ 0.15837032]], dtype=float32)>, <tf.Variable 'layer3/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 [26]:
```
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 [==============================] - 1s 1ms/step - loss: 0.6244
Epoch 2/20
32/32 [==============================] - 0s 935us/step - loss: 0.4763
Epoch 3/20
32/32 [==============================] - 0s 903us/step - loss: 0.3279
Epoch 4/20
32/32 [==============================] - 0s 871us/step - loss: 0.2236
Epoch 5/20
32/32 [==============================] - 0s 903us/step - loss: 0.1602
Epoch 6/20
32/32 [==============================] - 0s 903us/step - loss: 0.1224
Epoch 7/20
32/32 [==============================] - 0s 871us/step - loss: 0.0978
Epoch 8/20
32/32 [==============================] - 0s 871us/step - loss: 0.0811
Epoch 9/20
32/32 [==============================] - 0s 871us/step - loss: 0.0690
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)](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 [27]:
```
prediction = model.predict(X[0].reshape(1,400))  # a zero
print(f" predicting a zero: {prediction}")
prediction = model.predict(X[500].reshape(1,400))  # a one
print(f" predicting a one:  {prediction}")
```

```
1/1 [==============================] - 0s 71ms/step
 predicting a zero: [[0.02494038]]
1/1 [==============================] - 0s 15ms/step
 predicting a one:  [[0.98850954]]
```

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 [28]:
```
if prediction >= 0.5:
    yhat = 1
else:
    yhat = 0
print(f"prediction after threshold: {yhat}")
```

```
prediction after threshold: 1
```

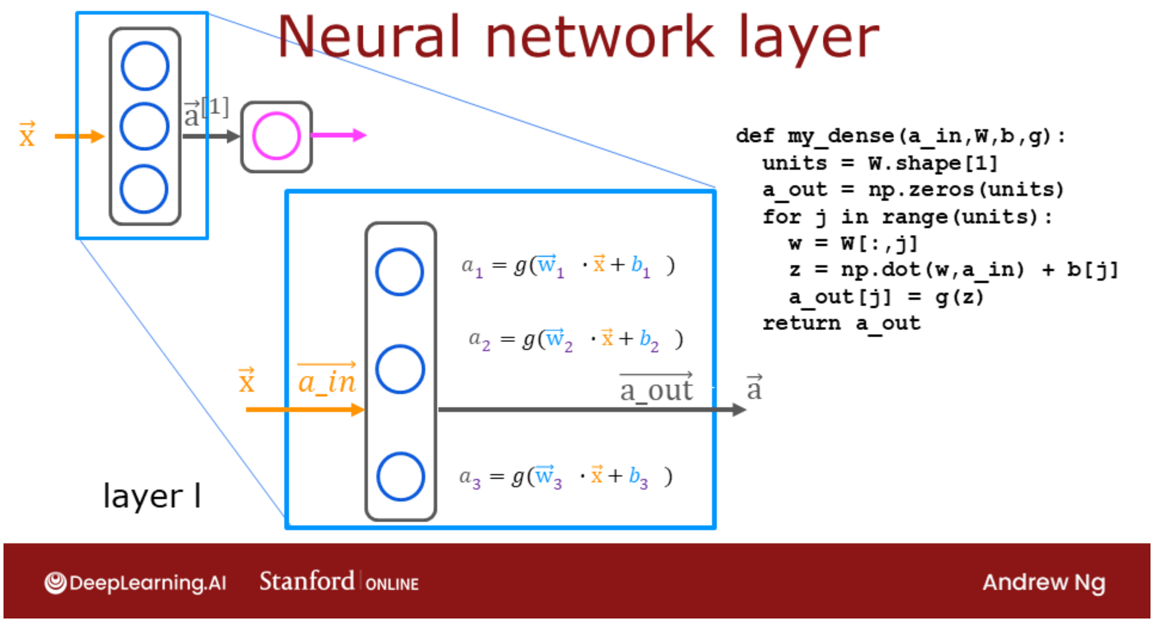
Let's compare the predictions vs the labels for a random sample of 64 digits. This takes a moment to run.

In [29]:
```
import warnings
warnings.simplefilter(action='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')

    # Predict using the Neural Network
    prediction = model.predict(X[random_index].reshape(1,400))
    if prediction >= 0.5:
        yhat = 1
    else:
        yhat = 0

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

```
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 16ms/step
```

## 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
$\vec{x}$
$\vec{a}^{[1]}$
$a_1 = g(\vec{w}_1 \cdot \vec{x} + b_1)$
$a_2 = g(\vec{w}_2 \cdot \vec{x} + b_2)$
$a_3 = g(\vec{w}_3 \cdot \vec{x} + b_3)$
$\vec{x}$
$\overrightarrow{a\_in}$
$\overrightarrow{a\_out}$
$\vec{a}$
layer l
DeepLearning.AI
Stanford ONLINE
Andrew Ng

```
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
```

## 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 [44]:
```
# 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 (units):
        w=W[:,j]
        z=np.dot(w,a_in)+b[j]
        a_out[j]=g(z)

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

```
In [45]: # 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]
```

```
In [46]: # UNIT TESTS
test_c2(my_dense)
```

```
All tests passed!
```

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

```
In [47]: 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 [48]: W1_tmp,b1_tmp = layer1.get_weights()
W2_tmp,b2_tmp = layer2.get_weights()
W3_tmp,b3_tmp = layer3.get_weights()
```

```
In [49]: # make predictions
prediction = my_sequential(X[0], W1_tmp, b1_tmp, W2_tmp, b2_tmp, W3_tmp, b3_tmp )
if prediction >= 0.5:
    yhat = 1
else:
    yhat = 0
print( "yhat = ", yhat, " label= ", y[0,0])
prediction = my_sequential(X[500], W1_tmp, b1_tmp, W2_tmp, b2_tmp, W3_tmp, b3_tmp )
if prediction >= 0.5:
    yhat = 1
else:
    yhat = 0
print( "yhat = ", yhat, " label= ", y[500,0])
```

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

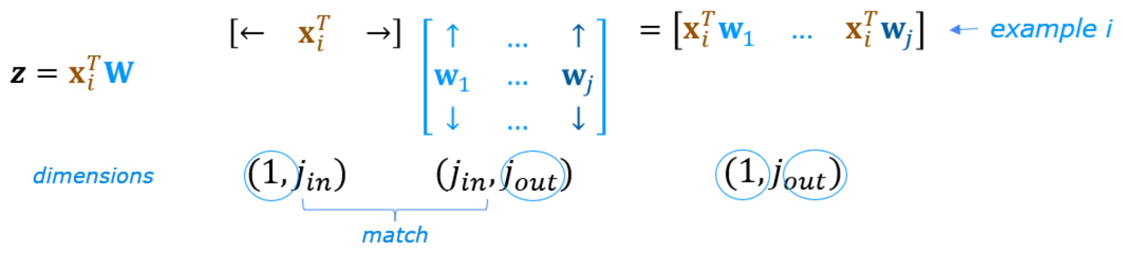
Run the following cell to see predictions from both the Numpy model and the Tensorflow model. This takes a moment to run.

In [50]:
```
import warnings
warnings.simplefilter(action='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')

    # Predict using the Neural Network implemented in Numpy
    my_prediction = my_sequential(X[random_index], W1_tmp, b1_tmp, W2_tmp, b2_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(f"{y[random_index,0]},{tf_yhat},{my_yhat}")
    ax.set_axis_off()
fig.suptitle("Label, yhat Tensorflow, yhat Numpy", fontsize=16)
plt.show()
```

```
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 16ms/step
```

## 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:



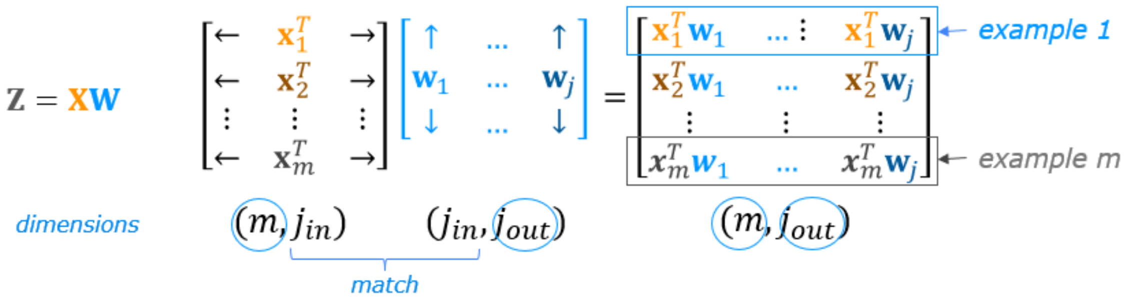
$z = \mathbf{x}_i^T\mathbf{W}$
$[\leftarrow \ \mathbf{x}_i^T \ \rightarrow]$
$\mathbf{w}_1 \ \dots \ \mathbf{w}_j$
$= [\mathbf{x}_i^T\mathbf{w}_1 \ \dots \ \mathbf{x}_i^T\mathbf{w}_j]$
example i
dimensions
$(1, j_{in})$
$(j_{in}, j_{out})$
$(1, j_{out})$
match

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 [51]:
```
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.



$\mathbf{Z} = \mathbf{X}\mathbf{W}$
$\mathbf{x}_1^T$
$\mathbf{x}_2^T$
$\mathbf{x}_m^T$
$\mathbf{w}_1 \ \dots \ \mathbf{w}_j$
$\mathbf{x}_1^T\mathbf{w}_1 \ \dots \ \mathbf{x}_1^T\mathbf{w}_j$
$\mathbf{x}_2^T\mathbf{w}_1 \ \dots \ \mathbf{x}_2^T\mathbf{w}_j$
$\mathbf{x}_m^T\mathbf{w}_1 \ \dots \ \mathbf{x}_m^T\mathbf{w}_j$
example 1
example m
dimensions
$(m, j_{in})$
$(j_{in}, j_{out})$
$(m, j_{out})$
match

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()` .

```
In [63]: # UNQ_C3
         # GRADED 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 (1,j)) : bias vector, j units
               g    activation function (e.g. sigmoid, relu..)
             Returns
               A_out (ndarray (m,j)) : 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 [64]: X_tst = 0.1*np.arange(1,9,1).reshape(4,2) # (4 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(1,3) # (1, 3 features)
         A_tst = my_dense_v(X_tst, W_tst, b_tst, sigmoid)
         print(A_tst)
```

```
tf.Tensor(
[[0.54735762 0.57932425 0.61063923]
 [0.57199613 0.61301418 0.65248946]
 [0.5962827  0.64565631 0.6921095 ]
 [0.62010643 0.67699586 0.72908792]], shape=(4, 3), dtype=float64)
```

**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]]
```

```
In [65]: # 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  [66]: 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  [67]: 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.

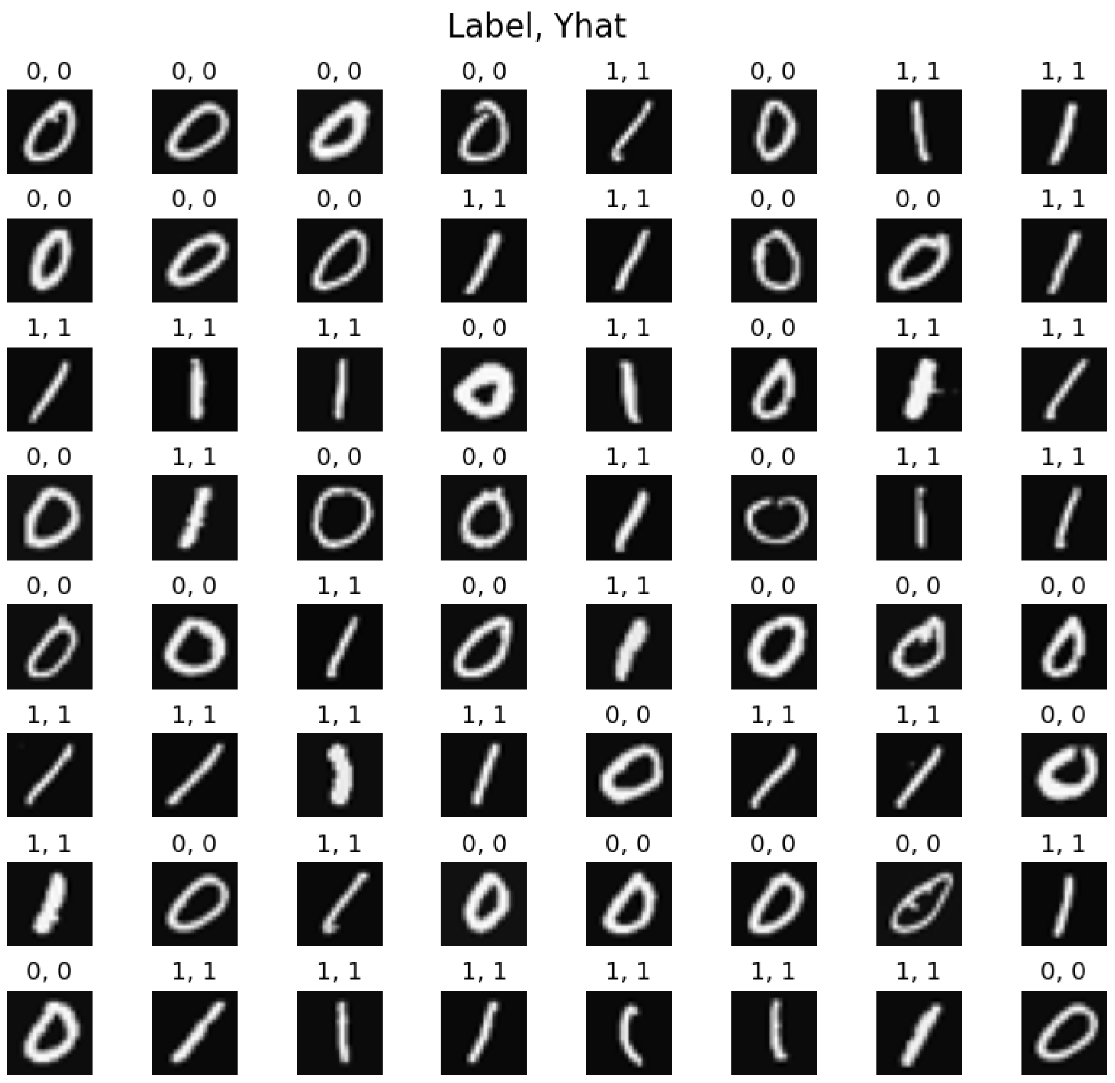
```
In  [68]: Prediction = my_sequential_v(X, W1_tmp, b1_tmp, W2_tmp, b2_tmp, W3_tmp, b3_tmp )
          Prediction.shape

Out[68]: TensorShape([1000, 1])
```

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

```
In  [69]: Yhat = (Prediction >= 0.5).numpy().astype(int)
          print("predict a zero: ",Yhat[0], "predict a one: ", Yhat[500])

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

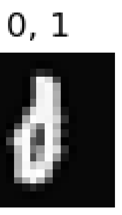
Run the following cell to see predictions. This will use the predictions we just calculated above. This takes a moment to run.

In [70]:
```
import warnings
warnings.simplefilter(action='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(f"{y[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  [71]: 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(f"{y[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, $\mathbf{Z} = \mathbf{XW} + \mathbf{b}$ 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): 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

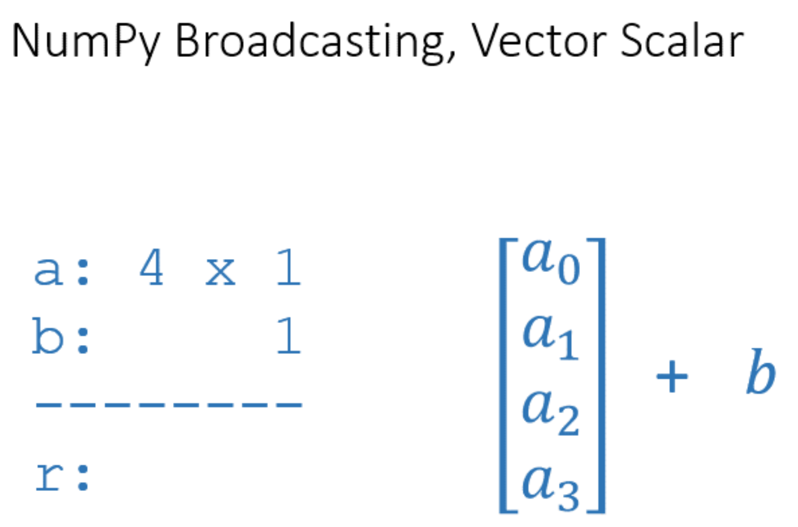
If these conditions are not met, a ValueError: operands could not be broadcast together exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the size that is not 1 along each axis of the inputs.

Here are some examples:

```
a:        m x 1      1 x n      4 x 1
b:            1          1      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:



NumPy Broadcasting, Vector Scalar

```
a:  4  x  1
b:        1
---------
r:
```

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} + b$$

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 [72]:
```python
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 [73]:
```python
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]]
```

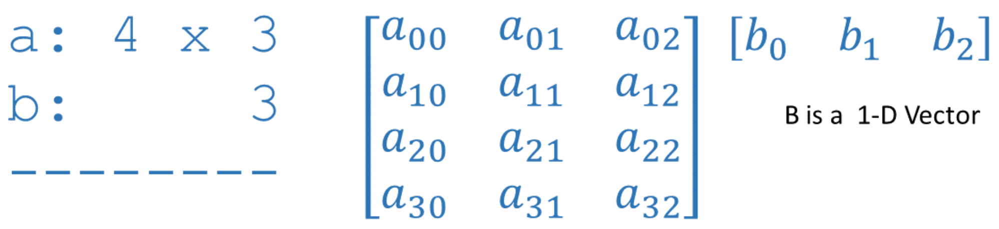
$$\boldsymbol{a} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} b_0 & b_1 & b_2 \end{bmatrix} \qquad a + b = b + a = \begin{array}{c|ccc} & [\ b_0 & b_1 & b_2\ ] \\ \hline \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} & \begin{bmatrix} a_0 + b_0 & a_0 + b_1 & a_0 + b_2 \\ a_1 + b_0 & a_1 + b_1 & a_1 + b_2 \\ a_2 + b_0 & a_2 + b_1 & a_2 + b_2 \\ a_3 + b_0 & a_3 + b_1 & a_3 + b_2 \end{bmatrix} \end{array}$$

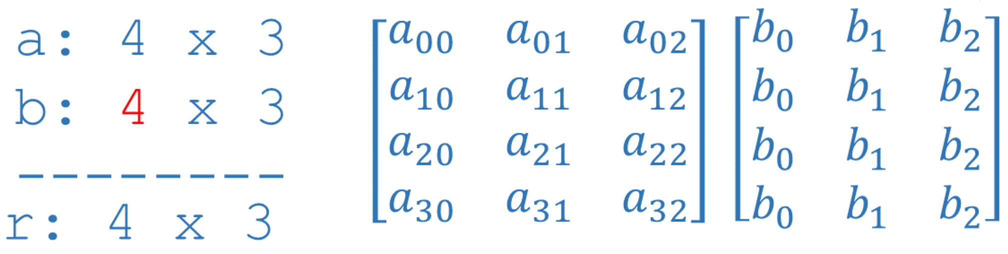
**Row-Column Element-Wise Operations**

In [74]:
```python
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 the dense layer you built above. Adding a 1-D vector $b$ to a (m,j) matrix.





## Matrix + 1-D Vector

In [ ]:

```

```

# 实验四 神经网络-多分类

## 一、实验目的

1. 掌握本地Jupyter notebook编译环境；
2. 熟悉Python编程集成环境Pychram、vs code；
3. 掌握基于numpy与Tensorflow库的神经网络模型搭建。

本实验代码于吴恩达老师Coursera中的机器学习专项课程。

## 二、实验内容

1. 安装并配置本地Jupyter notebook编译环境；
2. 基于numpy与Tensorflow库的神经网络多分类模型搭建。

## 三、实验结果

1. 完成本实验中的Exercise 1和2, 直到显示All tests passed!;

## 四、实验心得(500字以内)

实验心得：通过本次实验，我利用 Keras Sequential 模型和带有 ReLU 激活的 Dense Layer 构建了三层神经网络，用来实现手写体识别数字0-9。这三层网络的激活函数分别为'relu','relu','linear'。前两层采用relu函数不用sigmoid函数是因为sigmoid函数涉及e的乘方运算，计算量非常大，影响模型的效率。最后一层用linear函数不用softmax函数是因为softmax函数中也存在e的开放运算，由于float型变量存储位数有限，计算e的开方时可能存在溢出，结果存在累计误差，故采用linear函数。选择好损失函数和梯度下降策略后，进行40次epoch，就得到了0-9手写体识别的模型，然而在测试集上的测试结果其正确率并不是很理想。对比上一次实验简单的0-1识别只用了20次epoch，我猜测如果增加epoch模型应当更好。于是改换50次epoch重新测试，测试集全部通过。当然了，是不是过拟合，是不是一次偶然，希望在以后的学习中慢慢验证。

# Practice Lab: Neural Networks for Handwritten Digit Recognition, Multiclass

In this exercise, you will use a neural network to recognize the hand-written digits 0-9.

## Outline

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

In [182]:
```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.activations import linear, relu, sigmoid
import matplotlib.pyplot as plt
%matplotlib inline

plt.style.use('./deeplearning.mplstyle')

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

from public_tests import *

from autils import *
from lab_utils_softmax import plt_softmax
np.set_printoptions(precision=2)
```

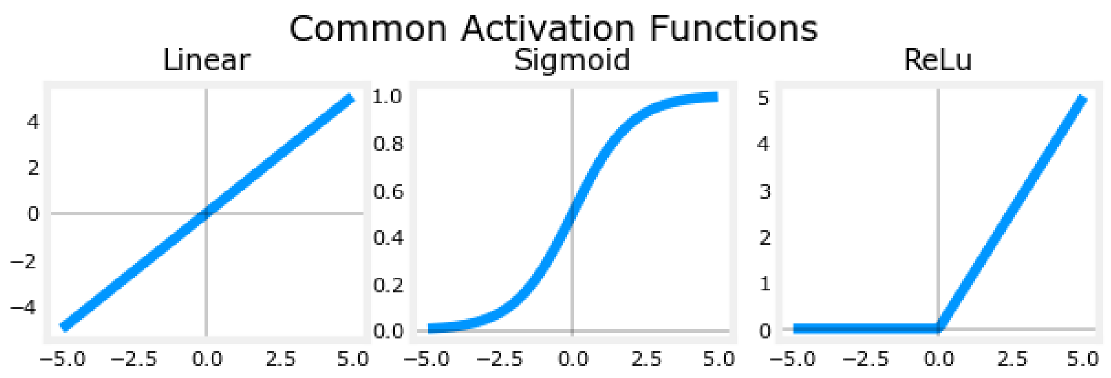
## 2 - ReLU Activation

This week, a new activation was introduced, the Rectified Linear Unit (ReLU).

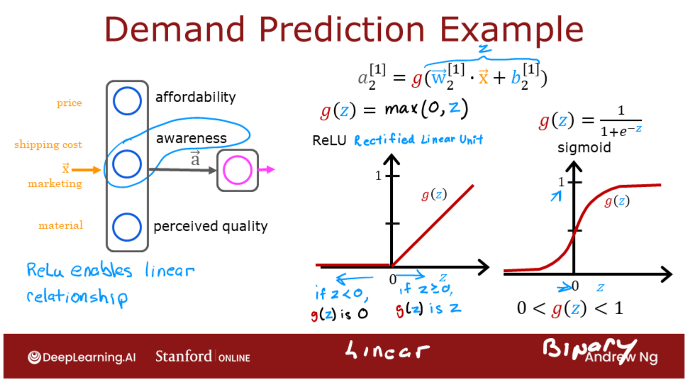
$$a = max(0, z) \quad\quad \text{\# ReLU function}$$

In [ ]:
```

```

In [183]:
```
plt_act_trio()
```

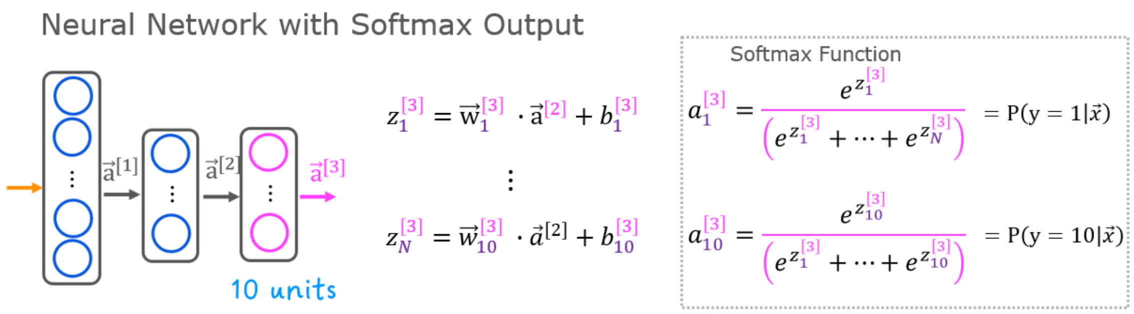


The example from the lecture on the right shows an application of the ReLU. In this example, the derived "awareness" feature is not binary but has a continuous range of values. The sigmoid is best for on/off or binary situations. The ReLU provides a continuous linear relationship. Additionally it has an 'off' range where the output is zero. The "off" feature makes the ReLU a Non-Linear activation. Why is this needed? This enables multiple units to contribute to to the resulting function without interfering. This is examined more in the supporting optional lab.



## 3 - Softmax Function

A multiclass neural network generates N outputs. One output is selected as the predicted answer. In the output layer, a vector $\mathbf{z}$ is generated by a linear function which is fed into a softmax function. The softmax function converts $\mathbf{z}$ into a probability distribution as described below. After applying softmax, each output will be between 0 and 1 and the outputs will sum to 1. They can be interpreted as probabilities. The larger inputs to the softmax will correspond to larger output probabilities.


Neural Network with Softmax Output
$\vec{a}^{[1]}$
$\vec{a}^{[2]}$
$\vec{a}^{[3]}$
10 units
$z_1^{[3]} = \vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]}$
$z_N^{[3]} = \vec{w}_{10}^{[3]} \cdot \vec{a}^{[2]} + b_{10}^{[3]}$
Softmax Function
$a_1^{[3]} = \frac{e^{z_1^{[3]}}}{\left(e^{z_1^{[3]}} + \cdots + e^{z_N^{[3]}}\right)} = P(y=1|\vec{x})$
$a_{10}^{[3]} = \frac{e^{z_{10}^{[3]}}}{\left(e^{z_1^{[3]}} + \cdots + e^{z_{10}^{[3]}}\right)} = P(y=10|\vec{x})$

The softmax function can be written:

$$a_j = \frac{e^{z_j}}{\sum_{k=0}^{N-1} e^{z_k}} \tag{1}$$

Where $z = \mathbf{w} \cdot \mathbf{x} + b$ and N is the number of feature/categories in the output layer.

## Exercise 1

Let's create a NumPy implementation:

```
In [184]: # UNQ_C1
          # GRADED CELL: my_softmax

          def my_softmax(z):
              """ Softmax converts a vector of values to a probability distribution.
              Args:
                z (ndarray (N,))  : input data, N features
              Returns:
                a (ndarray (N,))  : softmax of z
              """
              ### START CODE HERE ###
              e_z=np.exp(z)
              total=np.sum(e_z)
              a=e_z/total
              ### END CODE HERE ###
              return a
```

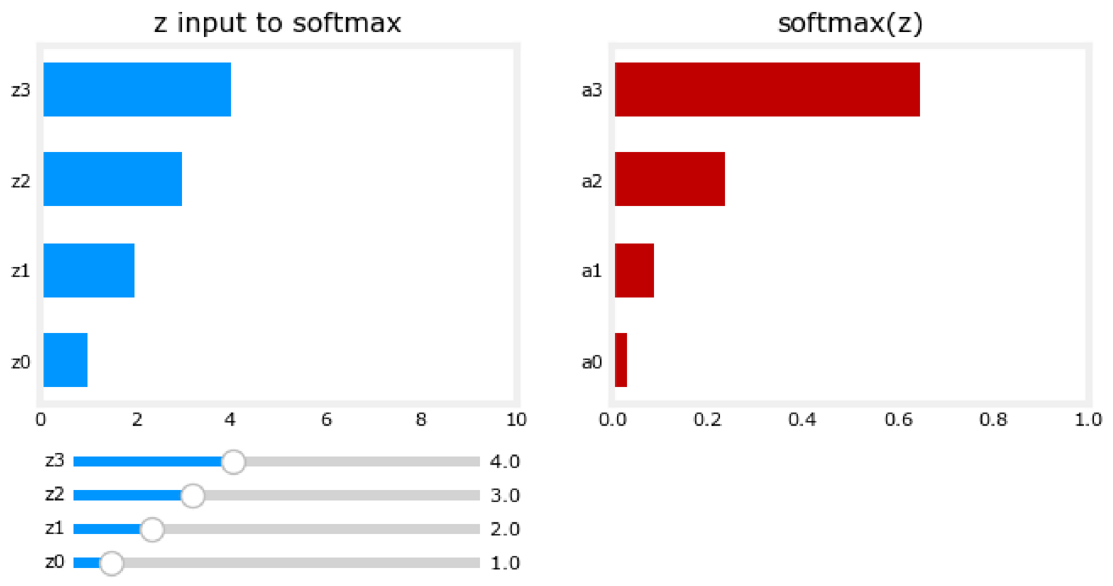
```
In [185]: z = np.array([1., 2., 3., 4.])
          a = my_softmax(z)
          atf = tf.nn.softmax(z)
          print(f"my_softmax(z):         {a}")
          print(f"tensorflow softmax(z): {atf}")

          # BEGIN UNIT TEST
          test_my_softmax(my_softmax)
          # END UNIT TEST
```

```
my_softmax(z):         [0.03 0.09 0.24 0.64]
tensorflow softmax(z): [0.03 0.09 0.24 0.64]
 All tests passed.
```

### Click for hints

Below, vary the values of the `z` inputs. Note in particular how the exponential in the numerator magnifies small differences in the values. Note as well that the output values sum to one.

In [186]:
```
plt.close("all")
plt_softmax(my_softmax)
```



## 4 - Neural Networks

In last weeks assignment, you implemented a neural network to do binary classification. This week you will extend that to multiclass classification. This will utilize the softmax activation.

### 4.1 Problem Statement

In this exercise, you will use a neural network to recognize ten handwritten digits, 0-9. This is a multiclass classification task where one of n choices is selected. Automated handwritten digit recognition is widely used today - from recognizing zip codes (postal codes) on mail envelopes to recognizing amounts written on bank checks.

### 4.2 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 5000 training examples of handwritten digits [1].
  - 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 examples becomes a single row in our data matrix `X` .
    - This gives us a 5000 x 400 matrix `X` where every row is a training example of a handwritten digit image.

$$X = \left(\begin{array}{cc} --- (x^{(1)}) --- \\ --- (x^{(2)}) --- \\ \vdots \\ --- (x^{(m)}) --- \end{array}\right)$$

- The second part of the training set is a 5000 x 1 dimensional vector `y` that contains labels for the training set
  - `y = 0` if the image is of the digit `0`, `y = 4` if the image is of the digit `4` and so on.

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

In [187]:
```
# load dataset
X, y = load_data()
```

### 4.2.1 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 the first element in the variables `X` and `y`.

In [188]:
```
print ('The first element of X is: ', X[0])
```

```
The first element of X is:  [ 0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+
00  0.00e+00  0.00e+00
  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
  0.00e+00  0.00e+00  0.00e+00  0.00e+00  8.56e-06  1.94e-06 -7.37e-04
 -8.13e-03 -1.86e-02 -1.87e-02 -1.88e-02 -1.91e-02 -1.64e-02 -3.78e-03
  3.30e-04  1.28e-05  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
  0.00e+00  0.00e+00  1.16e-04  1.20e-04 -1.40e-02 -2.85e-02  8.04e-02
  2.67e-01  2.74e-01  2.79e-01  2.74e-01  2.25e-01  2.78e-02 -7.06e-03
  2.35e-04  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00
  1.28e-17 -3.26e-04 -1.39e-02  8.16e-02  3.83e-01  8.58e-01  1.00e+00
  9.70e-01  9.31e-01  1.00e+00  9.64e-01  4.49e-01 -5.60e-03 -3.78e-03
  0.00e+00  0.00e+00  0.00e+00  0.00e+00  5.11e-06  4.36e-04 -3.96e-03
```

In [189]:
```
print ('The first element of y is: ', y[0,0])
print ('The last element of y is: ', y[-1,0])
```

```
The first element of y is:  0
The last element of y is:  9
```

### 4.2.2 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 [190]:
```python
print ('The shape of X is: ' + str(X.shape))
print ('The shape of y is: ' + str(y.shape))
```

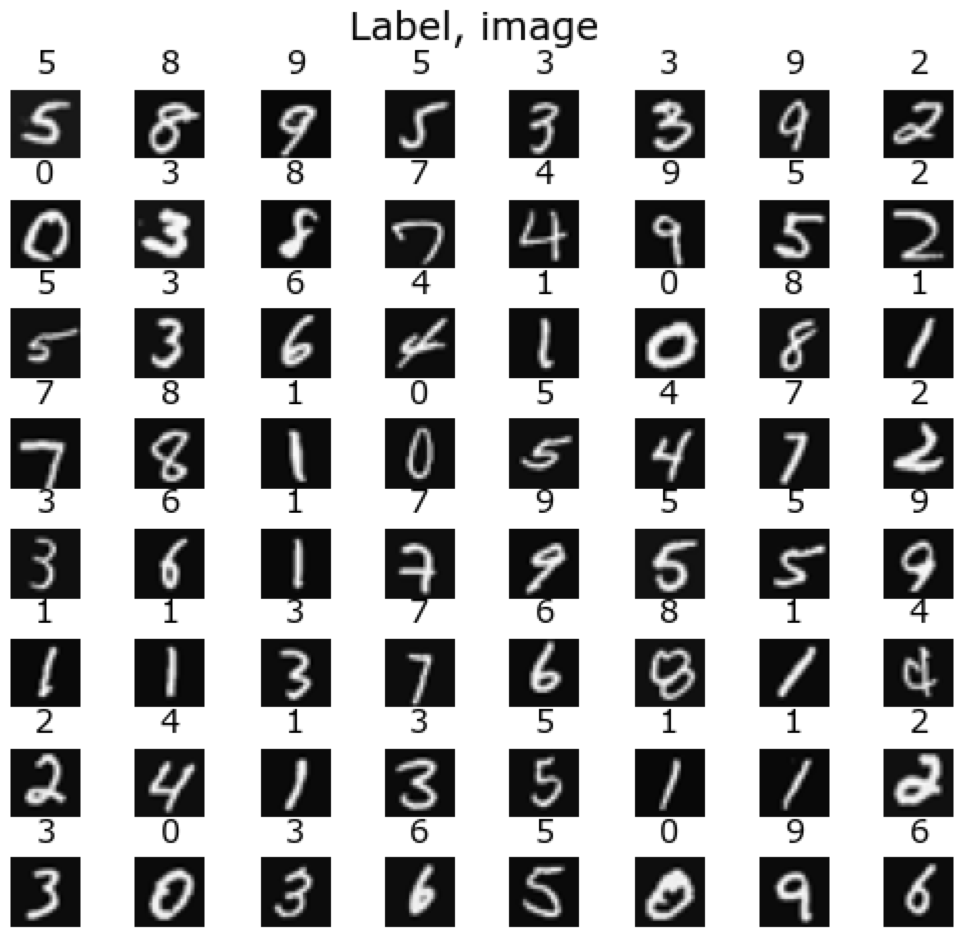
```
The shape of X is: (5000, 400)
The shape of y is: (5000, 1)
```

#### 4.2.3 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

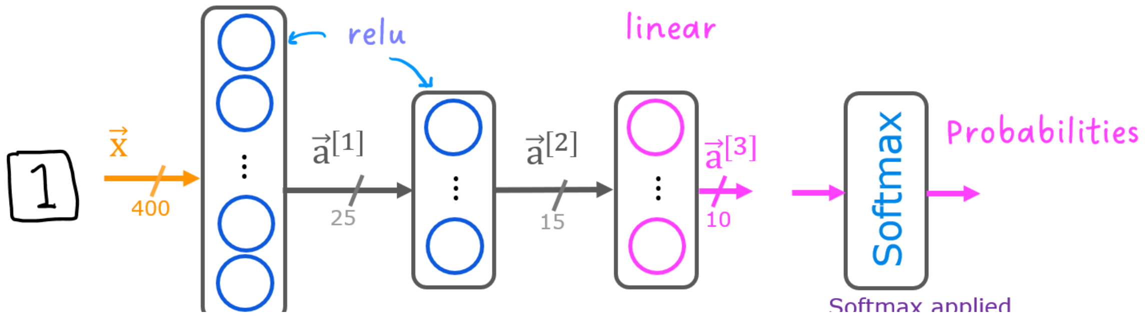
In [191]:
```python
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
# You do not need to modify anything in this cell

m, n = X.shape

fig, axes = plt.subplots(8,8, figsize=(5,5))
fig.tight_layout(pad=0.13,rect=[0, 0.03, 1, 0.91]) #[left, bottom, right, top]

#fig.tight_layout(pad=0.5)
widgvis(fig)
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(y[random_index,0])
    ax.set_axis_off()
    fig.suptitle("Label, image", fontsize=14)
```



## 4.3 Model representation

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

- This has two dense layers with ReLU activations followed by an output layer with a linear activation.
    - Recall that our inputs are pixel values of digit images.
    - Since the images are of size $20 \times 20$, 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 $10$ output units in layer 3, one for each digit.
    - Recall that the dimensions of these parameters is 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 be 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, 10) and the shape of `b3` is: (10,)

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

## 4.4 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.

## 4.5 Softmax placement

As described in the lecture and the optional softmax lab, numerical stability is improved if the softmax is grouped with the loss function rather than the output layer during training. This has implications when *building* the model and *using* the model.
Building:

- The final Dense layer should use a 'linear' activation. This is effectively no activation.
- The `model.compile` statement will indicate this by including `from_logits=True`.
  `loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)`
- This does not impact the form of the target. In the case of SparseCategorialCrossentropy, the target is the expected digit, 0-9.

Using the model:

- The outputs are not probabilities. If output probabilities are desired, apply a softmax

## Exercise 2

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

In [192]:
```
# UNQ_C2
# GRADED CELL: Sequential model
tf.random.set_seed(1234) # for consistent results
model = Sequential(
    [
        ### START CODE HERE ###
        tf.keras.Input(shape=(400,)),
        Dense(25,activation='relu',name='layer1'),
        Dense(15,activation='relu',name='layer2'),
        Dense(10,activation='linear',name='layer3')
        ### END CODE HERE ###
    ], name = "my_model"
)
```

In [193]:
```
model.summary()
```

```
Model: "my_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 layer1 (Dense)              (None, 25)                10025

 layer2 (Dense)              (None, 15)                390

 layer3 (Dense)              (None, 10)                160

=================================================================
Total params: 10575 (41.31 KB)
Trainable params: 10575 (41.31 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

## Expected Output (Click to expand)

In [194]:
```
# BEGIN UNIT TEST
test_model(model, 10, 400)
# END UNIT TEST
```

```
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.

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

In [195]:
```
[layer1, layer2, layer3] = model.layers
```

In [196]:
```
#### 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 = (400, 25), b1 shape = (25,)
W2 shape = (25, 15), b2 shape = (15,)
W3 shape = (15, 10), b3 shape = (10,)
```

**Expected Output**

```
W1 shape = (400, 25), b1 shape = (25,)
W2 shape = (25, 15), b2 shape = (15,)
W3 shape = (15, 10), b3 shape = (10,)
```

The following code:

- defines a loss function, `SparseCategoricalCrossentropy` and indicates the softmax should be included with the loss calculation by adding `from_logits=True` )
- defines an optimizer. A popular choice is Adaptive Moment (Adam) which was described in lecture.

```
In [197]: model.compile(
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
          )

          history = model.fit(
              X,y,
              epochs=50
          )
```

```
Epoch 1/50
157/157 [==============================] - 0s 980us/step - loss: 1.4973
Epoch 2/50
157/157 [==============================] - 0s 886us/step - loss: 0.6038
Epoch 3/50
157/157 [==============================] - 0s 910us/step - loss: 0.4151
Epoch 4/50
157/157 [==============================] - 0s 890us/step - loss: 0.3298
Epoch 5/50
157/157 [==============================] - 0s 895us/step - loss: 0.2806
Epoch 6/50
157/157 [==============================] - 0s 923us/step - loss: 0.2453
Epoch 7/50
157/157 [==============================] - 0s 888us/step - loss: 0.2175
Epoch 8/50
157/157 [==============================] - 0s 865us/step - loss: 0.1960
Epoch 9/50
157/157 [==============================] - 0s 853us/step - loss: 0.1830
Epoch 10/50
```

### Epochs and batches

In the `compile` statement above, the number of `epochs` was set to 100. This specifies that the entire data set should be applied during training 100 times. During training, you see output describing the progress of training that looks like this:

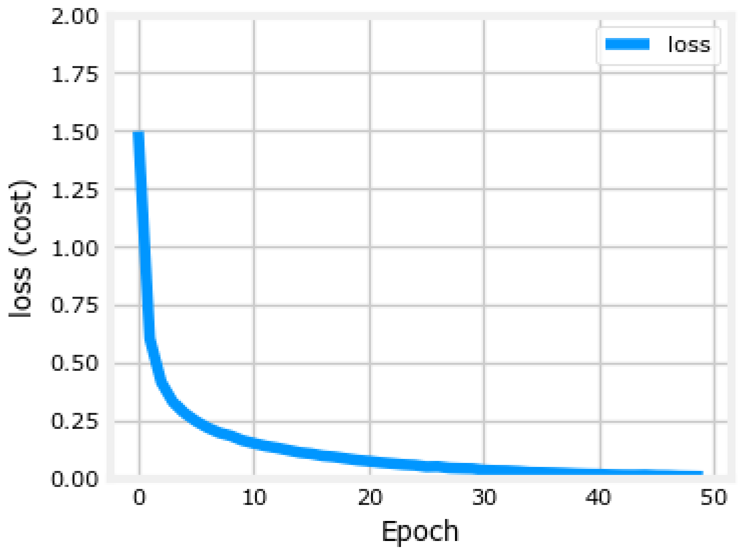
```
Epoch 1/100
157/157 [==============================] - 0s 1ms/step - loss: 2.2770
```

The first line, `Epoch 1/100`, describes which epoch the model is currently running. For efficiency, the training data set is broken into 'batches'. The default size of a batch in Tensorflow is 32. There are 5000 examples in our data set or roughly 157 batches. The notation on the 2nd line `157/157 [====` is describing which batch has been executed.

### Loss (cost)

In course 1, we learned to track the progress of gradient descent by monitoring the cost. Ideally, the cost will decrease as the number of iterations of the algorithm increases. Tensorflow refers to the cost as `loss`. Above, you saw the loss displayed each epoch as `model.fit` was executing. The [.fit (https://www.tensorflow.org/api_docs/python/tf/keras/Model)](https://www.tensorflow.org/api_docs/python/tf/keras/Model) method returns a variety of metrics including the loss. This is captured in the `history` variable above. This can be used to examine the loss in a plot as shown below.

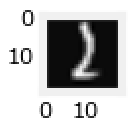
In [198]:
```
plot_loss_tf(history)
```



**Prediction**

To make a prediction, use Keras `predict`. Below, X[1015] contains an image of a two.

In [199]:
```
image_of_two = X[1015]
display_digit(image_of_two)

prediction = model.predict(image_of_two.reshape(1,400))  # prediction

print(f" predicting a Two: \n{prediction}")
print(f" Largest Prediction index: {np.argmax(prediction)}")
```



```
1/1 [==============================] - 0s 37ms/step
 predicting a Two:
[[ -7.04   3.76  12.26   7.94 -13.29  -5.39  -9.93   8.02   0.29  -4.86]]
 Largest Prediction index: 2
```

The largest output is prediction[2], indicating the predicted digit is a '2'. If the problem only requires a selection, that is sufficient. Use NumPy [argmax (https://numpy.org/doc/stable/reference/generated/numpy.argmax.html)](https://numpy.org/doc/stable/reference/generated/numpy.argmax.html) to select it. If the problem requires a probability, a softmax is required:

```
In  [200]: prediction_p = tf.nn.softmax(prediction)

           print(f" predicting a Two. Probability vector: \n{prediction_p}")
           print(f"Total of predictions: {np.sum(prediction_p):0.3f}")
```

```
 predicting a Two. Probability vector:
[[4.04e-09 1.98e-04 9.73e-01 1.30e-02 7.84e-12 2.11e-08 2.24e-10 1.41e-02
  6.19e-06 3.59e-08]]
Total of predictions: 1.000
```

To return an integer representing the predicted target, you want the index of the largest probability. This is accomplished with the Numpy argmax (https://numpy.org/doc/stable/reference/generated/numpy.argmax.html) function.

```
In  [201]: yhat = np.argmax(prediction_p)

           print(f"np.argmax(prediction_p): {yhat}")
```

```
np.argmax(prediction_p): 2
```

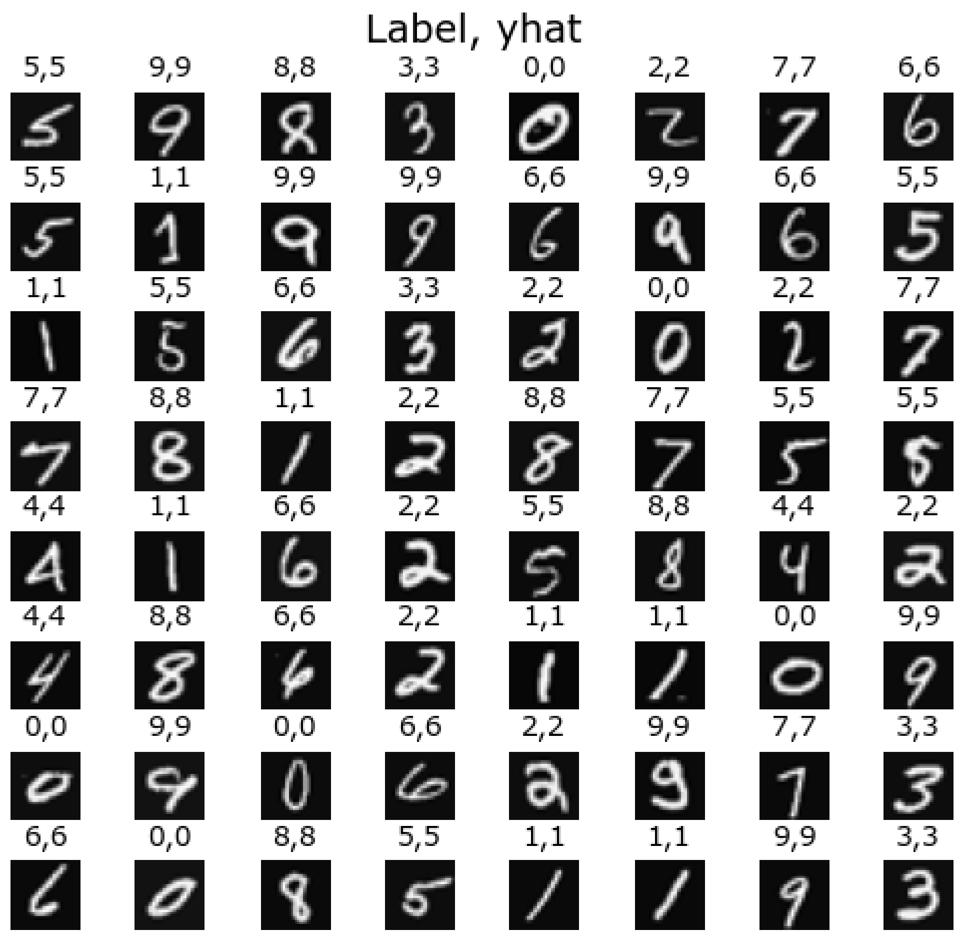
Let's compare the predictions vs the labels for a random sample of 64 digits. This takes a moment to run.

In [202]:

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

m, n = X.shape

fig, axes = plt.subplots(8,8, figsize=(5,5))
fig.tight_layout(pad=0.13,rect=[0, 0.03, 1, 0.91]) #[left, bottom, right, top]
widgvis(fig)
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')

    # Predict using the Neural Network
    prediction = model.predict(X[random_index].reshape(1,400))
    prediction_p = tf.nn.softmax(prediction)
    yhat = np.argmax(prediction_p)

    # Display the label above the image
    ax.set_title(f"{y[random_index,0]},{yhat}",fontsize=10)
    ax.set_axis_off()
fig.suptitle("Label, yhat", fontsize=14)
plt.show()
```

```
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 13ms/step
```

```
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 17ms/step
```



Let's look at some of the errors.

> Note: increasing the number of training epochs can eliminate the errors on this data set.

```
In [203]: print( f"{display_errors(model,X,y)} errors out of {len(X)} images")
```

```
157/157 [==============================] - 0s 846us/step
no errors found
0 errors out of 5000 images
```

## Congratulations!

You have successfully built and utilized a neural network to do multiclass classification.

```
In [ ]:
```