Machine Learning: Specialization in Deep Learning

Overview

Logistic Regression

Logistic regression is a learning algorithm used in a supervised learning problem when the output y are all either zero or one. The goal of logistic regression is to minimize the error between its predictions and training data.

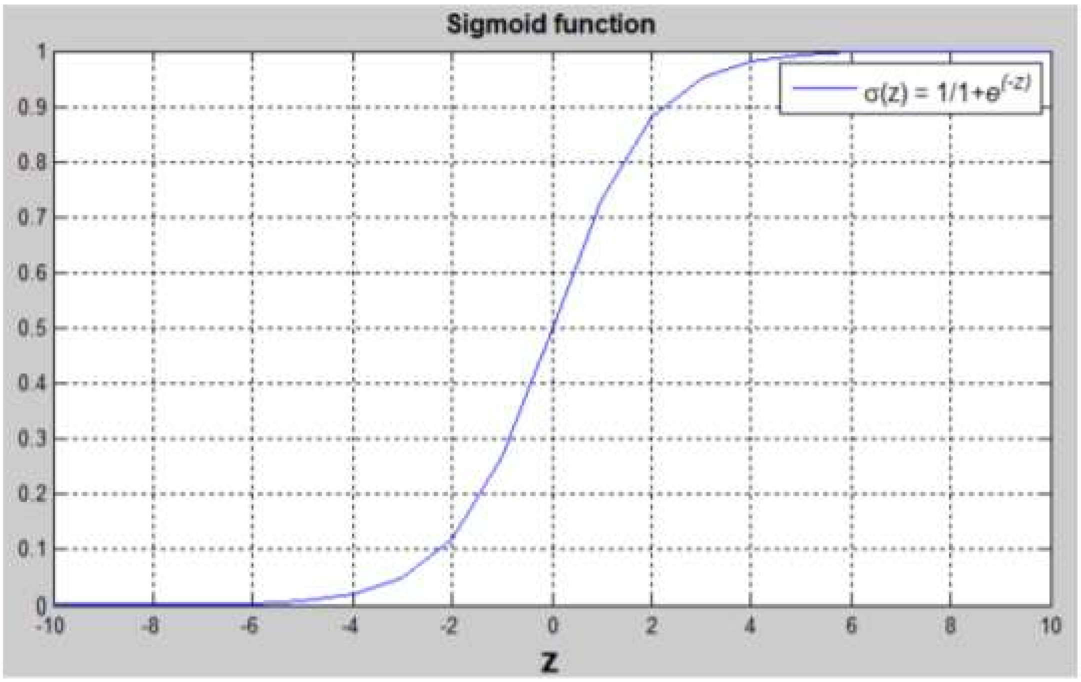
Example: Cat vs No - cat

Given an image which is represented by a feature vector x, the algorithm will evaluate the probability of a cat being in that image.

$Given\ x$, $\hat{y} = P(y = 1|x)$, where $0 \le \hat{y} \le 1$

The parameters used in Logistic regression are:

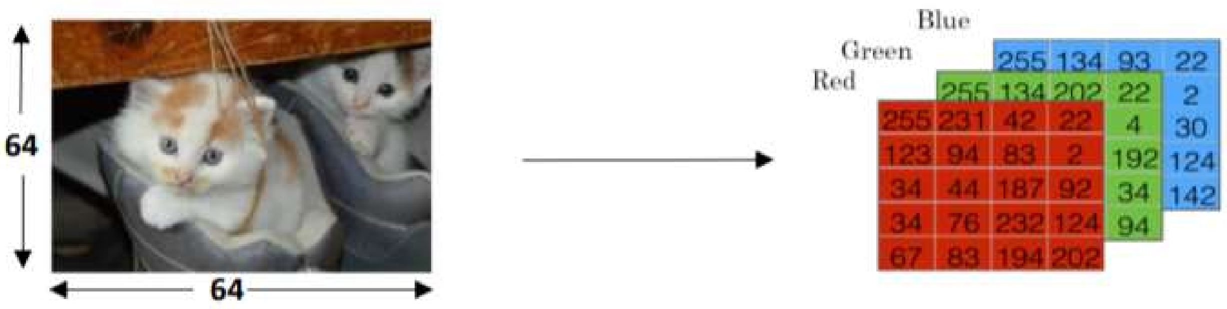
- The input features vector: $x \in \mathbb{R}^{nx}$, where nx is the number of features
- The training label: $y \in 0,1$
- The weights: $w \in \mathbb{R}^{nx}$, where nx is the number of features
- The threshold: $b \in \mathbb{R}$
- The output: $\hat{y}$ = sigmoid(z)
- Sigmoid function: s = $\sigma(w^T.x + b) = \sigma(z)$= 1/(1 + e$^{(-z)}$)



Binary Classsification

In a binary classification problem, the result is a discrete value output.

The goal is to train a classifier that the input is an image represented by a feature vector, x, and predicts whether the corresponding label y is 1 or 0. In this case, whether this is a cat image (1) or a non-cat image (0).



An image is store in the computer in three separate matrices corresponding to the Red, Green, and Blue color channels of the image. The three matrices have the same size as the image, for example, the resolution of the cat image is 64 pixels X 64 pixels, the three matrices (RGB) are 64 X 64 each.

The value in a cell represents the pixel intensity which will be used to create a feature vector of n-dimension. In pattern recognition and machine learning, a feature vector represents an object, in this case, a cat or no cat.

To create a feature vector, x, the pixel intensity values will be “unrolled” or “reshaped” for each color. The dimension of the input feature vector x is nx = 64 * 64 * 3 = 12,288.

$$x = \left.\begin{bmatrix} 255 \\ 231 \\ 42 \\ \vdots \\ 255 \\ 134 \\ 202 \\ \vdots \\ 255 \\ 134 \\ 93 \\ \vdots \end{bmatrix}\right. \begin{matrix} \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right] \text{red} \\ \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right] \text{green} \\ \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right] \text{blue} \end{matrix}$$

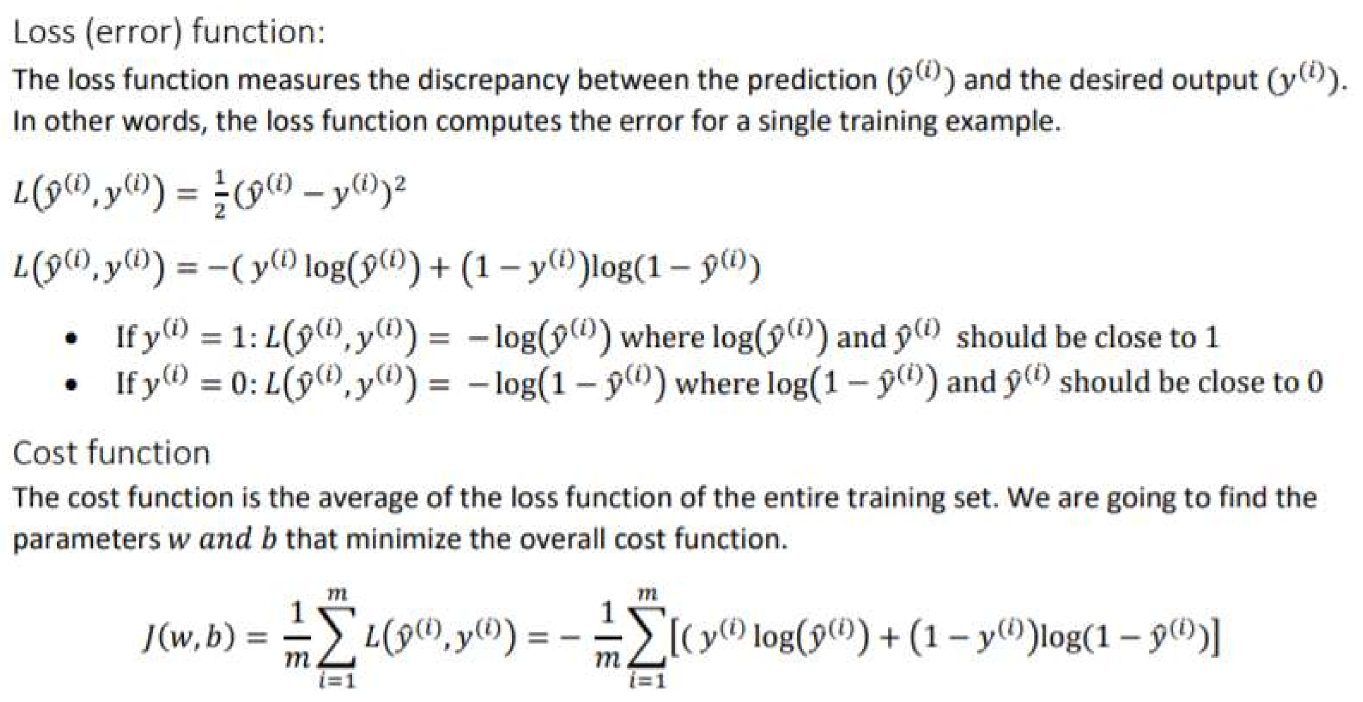
Cost and Loss Function

There are several ways to learn the parameters of a Machine Learning model.
We will focus on the approach that best illustrates statistical learning; minimising a cost function.

cost function—it helps the learner to correct / change behaviour to minimize mistakes.

Put simply, a cost function is a measure of how wrong the model is in terms of its ability to estimate the relationship between X and y.

Logistic Loss and Cost Function



Loss (error) function:

The loss function measures the discrepancy between the prediction ($\hat{y}^{(i)}$) and the desired output ($y^{(i)}$). In other words, the loss function computes the error for a single training example.

$$L(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2$$

$$L(\hat{y}^{(i)}, y^{(i)}) = -(\, y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)})\log(1 - \hat{y}^{(i)})$$

- If $y^{(i)} = 1$: $L(\hat{y}^{(i)}, y^{(i)}) = -\log(\hat{y}^{(i)})$ where $\log(\hat{y}^{(i)})$ and $\hat{y}^{(i)}$ should be close to 1
- If $y^{(i)} = 0$: $L(\hat{y}^{(i)}, y^{(i)}) = -\log(1 - \hat{y}^{(i)})$ where $\log(1 - \hat{y}^{(i)})$ and $\hat{y}^{(i)}$ should be close to 0

Cost function

The cost function is the average of the loss function of the entire training set. We are going to find the parameters $w \; and \; b$ that minimize the overall cost function.

$$J(w, b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[(\, y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)})\log(1 - \hat{y}^{(i)})]$$

In [10]:

```
#sigmoid function

import numpy as np # this means you can access numpy functions by writing np.function() ins

def sigmoid(x):
    """
    Compute the sigmoid of x

    Arguments:
    x -- A scalar or numpy array of any size

    Return:
    s -- sigmoid(x)
    """

    ### START CODE HERE ### (≈ 1 line of code)
    s = 1./(1.+np.exp(-1.*x))
    ### END CODE HERE ###

    return s
```

In [11]:

```
x = np.array([1, 2, 3])
sigmoid(x)
```

Out[11]:

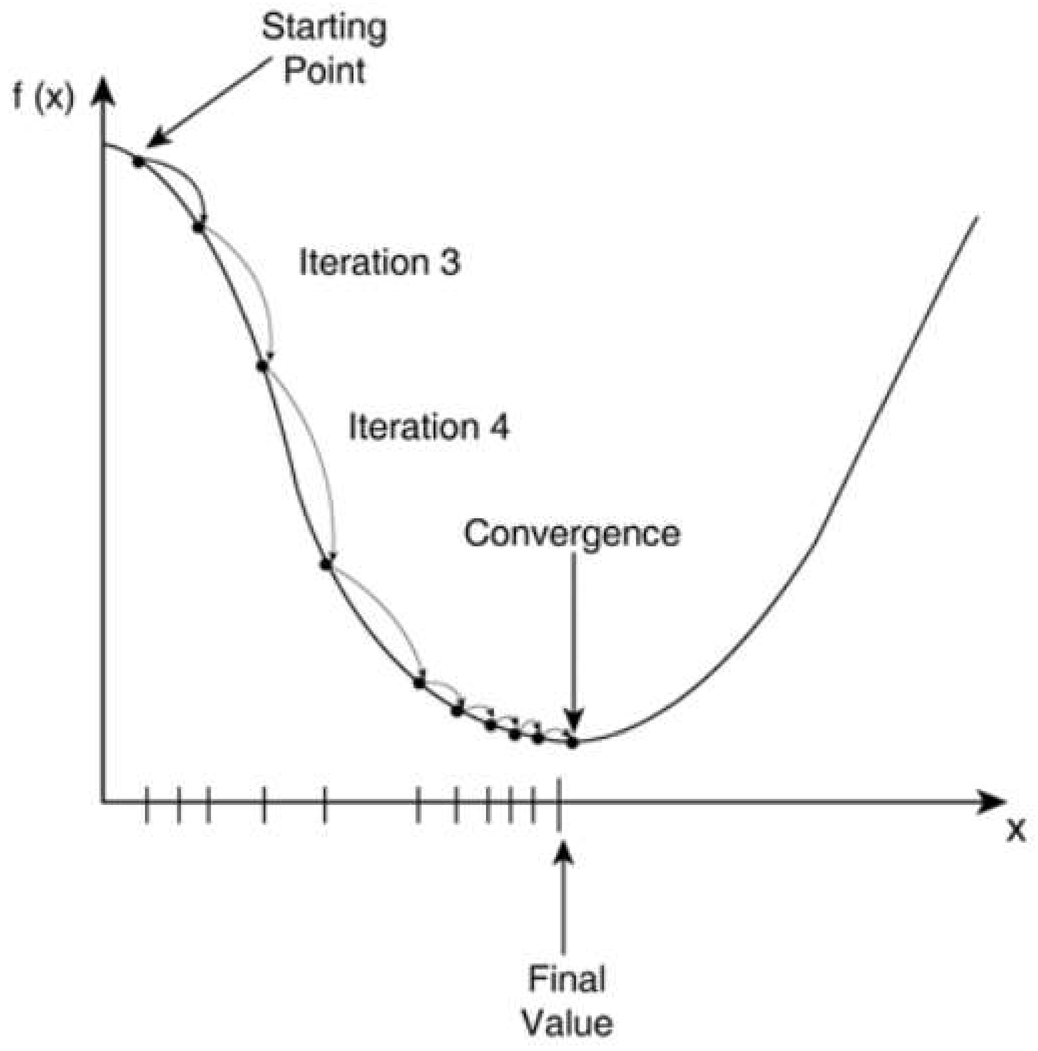
```
array([0.73105858, 0.88079708, 0.95257413])
```

Minimizing the cost function: Gradient descent

Now that we know that models learn by minimizing a cost function, you may naturally wonder how the cost function is minimized—enter gradient descent. Gradient descent is an efficient optimization algorithm that attempts to find a local or global minima of a function.

It can be simply called as Derivative.

Gradient descent enables a model to learn the gradient or direction that the model should take in order to reduce errors (differences between actual y and predicted y).



Type *Markdown* and LaTeX: α^2

Sigmoid Gradient

$$sigmoid\_derivative(x) = \sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{2}$$

You often code this function in two steps:

Set s to be the sigmoid of x. You might find your sigmoid(x) function useful. Compute $\sigma'(x) = s(1 - s)$

In [16]:

```
# sigmoid_derivative

def sigmoid_derivative(x):
    """
    Compute the gradient (also called the slope or derivative) of the sigmoid function with
    You can store the output of the sigmoid function into variables and then use it to calc

    Arguments:
    x -- A scalar or numpy array

    Return:
    ds -- Your computed gradient.
    """

    ### START CODE HERE ### (≈ 2 lines of code)
    s = sigmoid(x)
    ds = s*(1-s)
    ### END CODE HERE ###

    return ds
```

In [15]:

```
x = np.array([1, 2, 3])
print ("sigmoid_derivative(x) = " + str(sigmoid_derivative(x)))
```

```
sigmoid_derivative(x) = [0.19661193 0.10499359 0.04517666]
```

Activation Function

Logistic activation function in deep learning models, the activation function of a node defines the output of that node, or "neuron," given an input or set of inputs. This output is then used as input for the next node and so on until a desired solution to the original problem is found.

Softmax Activation Function

$$\phi(z) = \frac{e^{i}}{\sum_{j=0}^{k} e^{j}} \qquad \text{where i=0,1,....k}$$

In [20]:

```
def softmax(x):
    """Calculates the softmax for each row of the input x.

    Your code should work for a row vector and also for matrices of shape (n, m).

    Argument:
    x -- A numpy matrix of shape (n,m)

    Returns:
    s -- A numpy matrix equal to the softmax of x, of shape (n,m)
    """

    ### START CODE HERE ### (≈ 3 lines of code)
    # Apply exp() element-wise to x. Use np.exp(...).
    x_exp = np.exp(x)

    # Create a vector x_sum that sums each row of x_exp. Use np.sum(..., axis = 1, keepdims
    x_sum = np.sum(x_exp, axis=1, keepdims=True)

    # Compute softmax(x) by dividing x_exp by x_sum. It should automatically use numpy broa
    s = x_exp/x_sum

    ### END CODE HERE ###

    return s
```

In [21]:

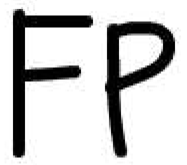
```
x = np.array([
    [9, 2, 5, 0, 0],
    [7, 5, 0, 0 ,0]])
print("softmax(x) = " + str(softmax(x)))
```

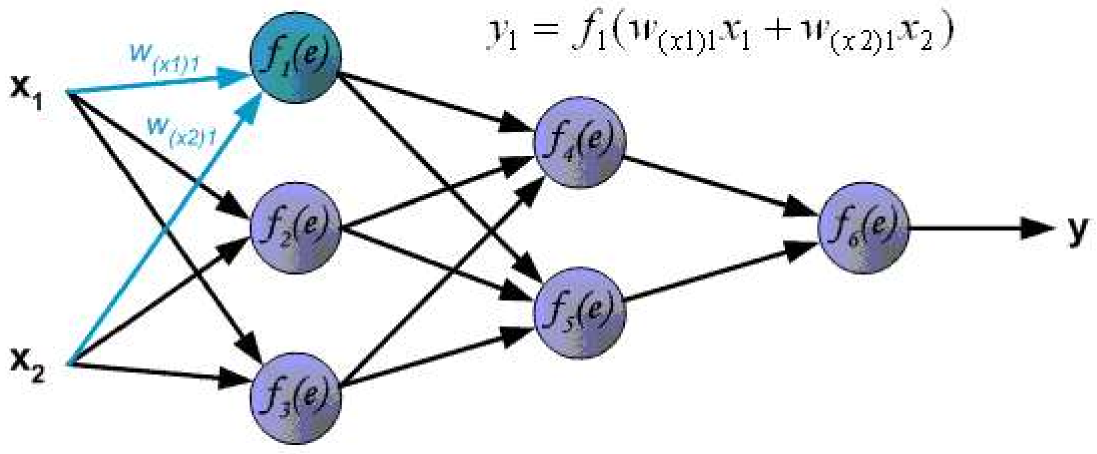
```
softmax(x) = [[9.80897665e-01 8.94462891e-04 1.79657674e-02 1.21052389e-04
  1.21052389e-04]
 [8.78679856e-01 1.18916387e-01 8.01252314e-04 8.01252314e-04
  8.01252314e-04]]
```

Forward and Backward propagation

In neural networks, you forward propagate to get the output and compare it with the real value to get the error.

Now, to minimize the error, you propagate backwards by finding the derivative of error with respect to each weight and then subtracting this value from the weight value. (For more read this (https://www.quora.com/What-is-the-difference-between-back-propagation-and-forward-propagation))





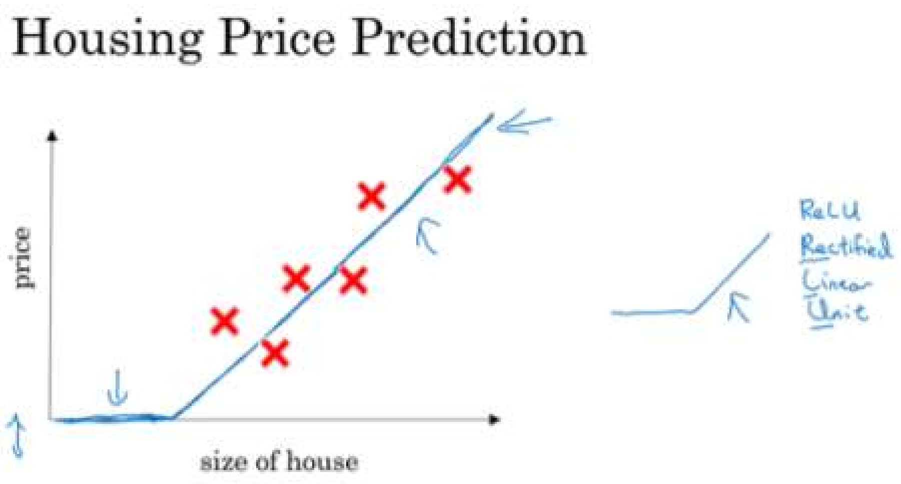
Shallow Neural Network

It is a powerful learning algorithm inspired by how the brain works.

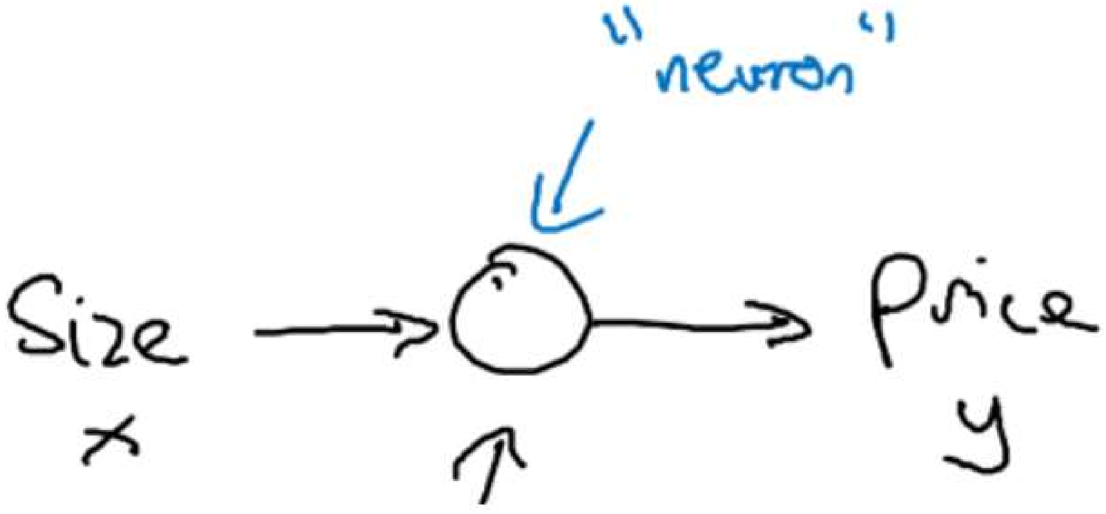
Example – single neural network

Given data about the size of houses on the real estate market and you want to fit a function that will predict their price. The function tries to solve the relationship between the size and the price of houses.

It is a linear regression problem because the price as a function of size is a continuous output. We know the prices can never be negative so we are creating a function called Rectified Linear Unit (ReLU) which starts at zero.



- The input is the size of the house (x)
- The output is the price (y)
- The training data (x, y) is plotted and the model predicts the relationship or the line between them.



Here the “neuron” implements the function ReLU (blue line)

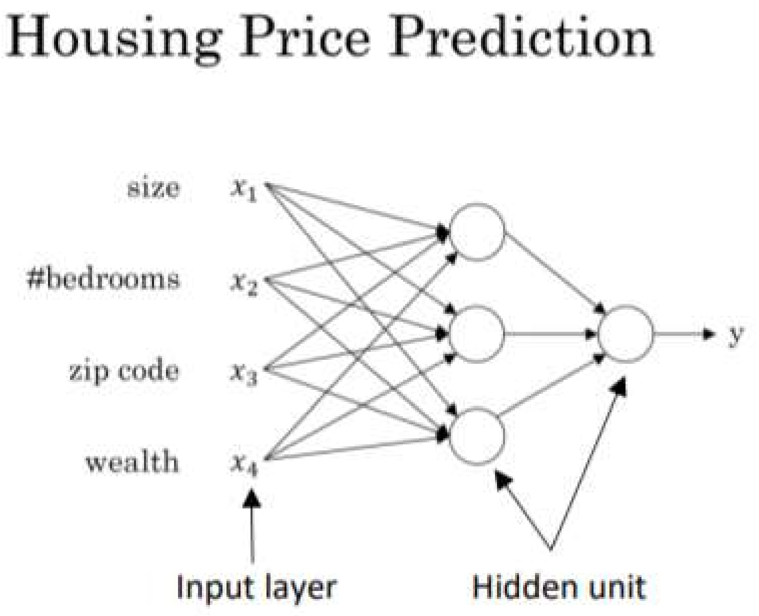
Deep Neural Network

Example – Multiple neural network

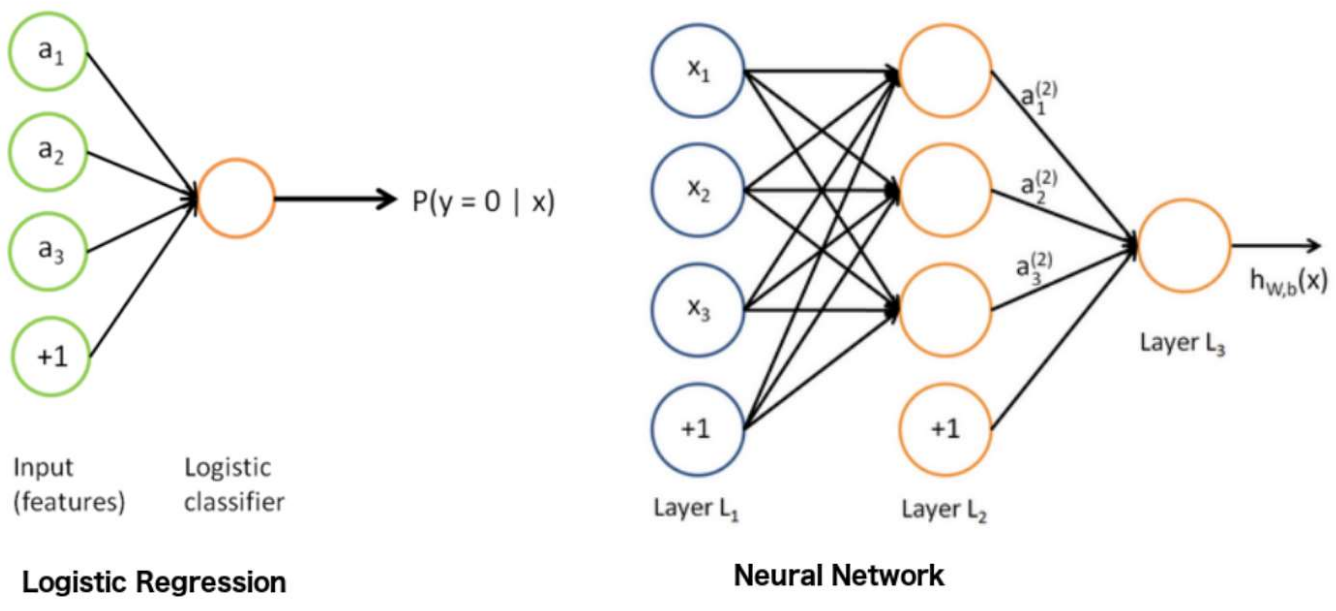
But in reality the price of a house can be affected by other features such as

- size
- number of bedrooms
- zip code
- wealth.

The role of the neural network is to predicted the price (red cross) and it will automatically generate the hidden units. We only need to give the inputs x and the output y.



Difference



Predicting house prices example with Keras

Importing Keras

In [1]:

```
import keras
keras.__version__
```

```
Using TensorFlow backend.
```

Out[1]:

```
'2.2.4'
```

The Boston Housing Price dataset

We will be attempting to predict the median price of homes in a given Boston suburb in the mid-1970s, given a few data points about the suburb at the time, such as the crime rate, the local property tax rate, etc.

The dataset we will be using has another interesting thing: it has very few data points, only 506 in total, split between 404 training samples and 102 test samples, and each "feature" in the input data (e.g. the crime rate is a feature) has a different scale. For instance some values are proportions, which take a values between 0 and 1, others take values between 1 and 12, others between 0 and 100...

Let's take a look at the data:

In [2]:

```
from keras.datasets import boston_housing

(train_data, train_targets), (test_data, test_targets) =  boston_housing.load_data()
```

In [3]:

```
train_data.shape
```

Out[3]:

```
(404, 13)
```

In [4]:

```
test_data.shape
```

Out[4]:

```
(102, 13)
```

In [5]:

```
train_targets
```

Out[5]:

```
array([15.2, 42.3, 50. , 21.1, 17.7, 18.5, 11.3, 15.6, 15.6, 14.4, 12.1,
       17.9, 23.1, 19.9, 15.7,  8.8, 50. , 22.5, 24.1, 27.5, 10.9, 30.8,
       32.9, 24. , 18.5, 13.3, 22.9, 34.7, 16.6, 17.5, 22.3, 16.1, 14.9,
       23.1, 34.9, 25. , 13.9, 13.1, 20.4, 20. , 15.2, 24.7, 22.2, 16.7,
       12.7, 15.6, 18.4, 21. , 30.1, 15.1, 18.7,  9.6, 31.5, 24.8, 19.1,
       22. , 14.5, 11. , 32. , 29.4, 20.3, 24.4, 14.6, 19.5, 14.1, 14.3,
       15.6, 10.5,  6.3, 19.3, 19.3, 13.4, 36.4, 17.8, 13.5, 16.5,  8.3,
       14.3, 16. , 13.4, 28.6, 43.5, 20.2, 22. , 23. , 20.7, 12.5, 48.5,
       14.6, 13.4, 23.7, 50. , 21.7, 39.8, 38.7, 22.2, 34.9, 22.5, 31.1,
       28.7, 46. , 41.7, 21. , 26.6, 15. , 24.4, 13.3, 21.2, 11.7, 21.7,
       19.4, 50. , 22.8, 19.7, 24.7, 36.2, 14.2, 18.9, 18.3, 20.6, 24.6,
       18.2,  8.7, 44. , 10.4, 13.2, 21.2, 37. , 30.7, 22.9, 20. , 19.3,
       31.7, 32. , 23.1, 18.8, 10.9, 50. , 19.6,  5. , 14.4, 19.8, 13.8,
       19.6, 23.9, 24.5, 25. , 19.9, 17.2, 24.6, 13.5, 26.6, 21.4, 11.9,
       22.6, 19.6,  8.5, 23.7, 23.1, 22.4, 20.5, 23.6, 18.4, 35.2, 23.1,
       27.9, 20.6, 23.7, 28. , 13.6, 27.1, 23.6, 20.6, 18.2, 21.7, 17.1,
        8.4, 25.3, 13.8, 22.2, 18.4, 20.7, 31.6, 30.5, 20.3,  8.8, 19.2,
       19.4, 23.1, 23. , 14.8, 48.8, 22.6, 33.4, 21.1, 13.6, 32.2, 13.1,
       23.4, 18.9, 23.9, 11.8, 23.3, 22.8, 19.6, 16.7, 13.4, 22.2, 20.4,
       21.8, 26.4, 14.9, 24.1, 23.8, 12.3, 29.1, 21. , 19.5, 23.3, 23.8,
       17.8, 11.5, 21.7, 19.9, 25. , 33.4, 28.5, 21.4, 24.3, 27.5, 33.1,
       16.2, 23.3, 48.3, 22.9, 22.8, 13.1, 12.7, 22.6, 15. , 15.3, 10.5,
       24. , 18.5, 21.7, 19.5, 33.2, 23.2,  5. , 19.1, 12.7, 22.3, 10.2,
       13.9, 16.3, 17. , 20.1, 29.9, 17.2, 37.3, 45.4, 17.8, 23.2, 29. ,
       22. , 18. , 17.4, 34.6, 20.1, 25. , 15.6, 24.8, 28.2, 21.2, 21.4,
       23.8, 31. , 26.2, 17.4, 37.9, 17.5, 20. ,  8.3, 23.9,  8.4, 13.8,
        7.2, 11.7, 17.1, 21.6, 50. , 16.1, 20.4, 20.6, 21.4, 20.6, 36.5,
        8.5, 24.8, 10.8, 21.9, 17.3, 18.9, 36.2, 14.9, 18.2, 33.3, 21.8,
       19.7, 31.6, 24.8, 19.4, 22.8,  7.5, 44.8, 16.8, 18.7, 50. , 50. ,
       19.5, 20.1, 50. , 17.2, 20.8, 19.3, 41.3, 20.4, 20.5, 13.8, 16.5,
       23.9, 20.6, 31.5, 23.3, 16.8, 14. , 33.8, 36.1, 12.8, 18.3, 18.7,
       19.1, 29. , 30.1, 50. , 50. , 22. , 11.9, 37.6, 50. , 22.7, 20.8,
       23.5, 27.9, 50. , 19.3, 23.9, 22.6, 15.2, 21.7, 19.2, 43.8, 20.3,
       33.2, 19.9, 22.5, 32.7, 22. , 17.1, 19. , 15. , 16.1, 25.1, 23.7,
       28.7, 37.2, 22.6, 16.4, 25. , 29.8, 22.1, 17.4, 18.1, 30.3, 17.5,
       24.7, 12.6, 26.5, 28.7, 13.3, 10.4, 24.4, 23. , 20. , 17.8,  7. ,
       11.8, 24.4, 13.8, 19.4, 25.2, 19.4, 19.4, 29.1])
```

Preparing the data

It would be problematic to feed into a neural network values that all take wildly different ranges. The network might be able to automatically adapt to such heterogeneous data, but it would definitely make learning more difficult. A widespread best practice to deal with such data is to do feature-wise normalization: for each feature in the input data (a column in the input data matrix), we will subtract the mean of the feature and divide by the standard deviation, so that the feature will be centered around 0 and will have a unit standard deviation. This is easily done in Numpy:

In [6]:

```
mean = train_data.mean(axis=0)
train_data -= mean
std = train_data.std(axis=0)
train_data /= std

test_data -= mean
test_data /= std
```

Building our network

Because so few samples are available, we will be using a very small network with two hidden layers, each with 64 units. In general, the less training data you have, the worse overfitting will be, and using a small network is one way to mitigate overfitting.

In [7]:

```
from keras import models
from keras import layers

def build_model():
    # Because we will need to instantiate
    # the same model multiple times,
    # we use a function to construct it.
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu',
                           input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

In [9]:

```
model = build_model()
model.fit(train_data, train_targets, validation_split= 0.1,
                        epochs=8, batch_size=1)
```

```
Train on 363 samples, validate on 41 samples
Epoch 1/8
363/363 [==============================] - 3s 8ms/step - loss: 166.6200 - me
an_absolute_error: 8.8905 - val_loss: 13.0517 - val_mean_absolute_error: 3.0
867
Epoch 2/8
363/363 [==============================] - 3s 9ms/step - loss: 23.4310 - mea
n_absolute_error: 3.0840 - val_loss: 9.3577 - val_mean_absolute_error: 2.366
9
Epoch 3/8
363/363 [==============================] - 2s 7ms/step - loss: 18.8193 - mea
n_absolute_error: 2.7602 - val_loss: 9.3822 - val_mean_absolute_error: 2.470
1
Epoch 4/8
363/363 [==============================] - 2s 7ms/step - loss: 15.7043 - mea
n_absolute_error: 2.6334 - val_loss: 8.0572 - val_mean_absolute_error: 2.314
5
Epoch 5/8
363/363 [==============================] - 2s 6ms/step - loss: 14.8218 - mea
n_absolute_error: 2.5479 - val_loss: 9.3294 - val_mean_absolute_error: 2.584
6
Epoch 6/8
363/363 [==============================] - 2s 6ms/step - loss: 13.2216 - mea
n_absolute_error: 2.4393 - val_loss: 7.7387 - val_mean_absolute_error: 2.361
4
Epoch 7/8
363/363 [==============================] - 3s 7ms/step - loss: 12.4759 - mea
n_absolute_error: 2.3421 - val_loss: 7.4302 - val_mean_absolute_error: 2.142
6
Epoch 8/8
363/363 [==============================] - 2s 7ms/step - loss: 12.1029 - mea
n_absolute_error: 2.3153 - val_loss: 6.6676 - val_mean_absolute_error: 1.997
8
```

Out[9]:

```
<keras.callbacks.History at 0x1dea2b704a8>
```

In [71]:

```
predict = model.predict( test_data[:10])
```

In [85]:

```
print("Predicted Output:")
print(predict)
print()
print("Ground Truth:")
print(test_targets[:10].reshape(10,1))
```

```
Predicted Output:
[[ 7.803312]
 [18.738247]
 [21.229338]
 [31.537516]
 [25.278505]
 [19.451117]
 [26.931568]
 [22.709377]
 [20.327257]
 [21.936008]]

Ground Truth:
[[ 7.2]
 [18.8]
 [19. ]
 [27. ]
 [22.2]
 [24.5]
 [31.2]
 [22.9]
 [20.5]
 [23.2]]
```

In [ ]:

```

```