Regression with Perceptron

In the week 2 assignment, you implemented the gradient descent method to build a linear regression model, predicting sales given a TV marketing budget. In this lab, you will construct a neural network corresponding to the same simple linear regression model. Then you will train the network, implementing the gradient descent method. After that you will increase the complexity of the neural network to build a multiple linear regression model, predicting house prices based on their size and quality.

Note: The same models were discussed in Course 1 "Linear Algebra" week 3 assignment, but model training with backward propagation was omitted.

Table of Contents

Packages

Let's first import all the required packages.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
# A library for data manipulation and analysis.
import pandas as pd

# Output of plotting commands is displayed inline within the Jupyter notebook.
%matplotlib inline

# Set a seed so that the results are consistent.
np.random.seed(3)
```

1 - Simple Linear Regression

1.1 - Simple Linear Regression Model

You can describe a simple linear regression model as

$$\hat{y} = wx + b,$$

where $\hat{y}$ is a prediction of dependent variable y based on independent variable x using a line equation with the slope w and intercept b.

Given a set of training data points (x_1, y_1), ..., (x_m, y_m), you will find the "best" fitting line - such parameters w and b that the differences between original values y_i and predicted values $\hat{y}_i = wx_i + b$ are minimum.

1.2 - Neural Network Model with a Single Perceptron and One Input Node

The simplest neural network model that describes the above problem can be realized by using one **perceptron**. The **input** and **output** layers will have one **node** each (x for input and $\hat{y} = z$ for output):

Weight (w) and **bias** (b) are the parameters that will get updated when you **train** the model. They are initialized to some random values or set to 0 and updated as the training progresses.

For each training example $x^{(i)}$, the prediction $\hat{y}^{(i)}$ can be calculated as:

$$z^{(i)} = wx^{(i)} + b,$$
$$\hat{y}^{(i)} = z^{(i)},$$

where $i = 1, \dots, m$.

You can organise all training examples as a vector X of size $(1 \times m)$ and perform scalar multiplication of X ($1 \times m$) by a scalar w, adding b, which will be broadcasted to a vector of size $(1 \times m)$:

$$Z = wX + b,$$
$$\hat{Y} = Z,$$

This set of calculations is called **forward propagation**.

For each training example you can measure the difference between original values $y^{(i)}$ and predicted values $\hat{y}^{(i)}$ with the **loss function** $L(w, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$. Division by 2 is taken just for scaling purposes, you will see the reason below, calculating partial derivatives. To compare the resulting vector of the predictions $\hat{Y}$ ($1 \times m$) with the vector Y of original values $y^{(i)}$, you can take an average of the loss function values for each of the training examples:

$$\mathcal{L}(w, b) = \frac{1}{2m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2.$$

This function is called the sum of squares **cost function**. The aim is to optimize the cost function during the training, which will minimize the differences between original values $y^{(i)}$ and predicted values $\hat{y}^{(i)}$.

When your weights were just initialized with some random values, and no training was done yet, you can't expect good results. You need to calculate the adjustments for the weight and bias, minimizing the cost function. This process is called **backward propagation**.

According to the gradient descent algorithm, you can calculate partial derivatives as:

$$\begin{align}
\frac{\partial \mathcal{L} }{ \partial w } &= \frac{1}{m}\sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right)x^{(i)},\\
\frac{\partial \mathcal{L} }{ \partial b } &= \frac{1}{m}\sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right).
\end{align}$$

You can see how the additional division by 2 in the equation (4) helped to simplify the results of the partial derivatives. Then update the parameters iteratively using the expressions

$$\begin{align}
w &= w - \alpha \frac{\partial \mathcal{L} }{ \partial w },\\
b &= b - \alpha \frac{\partial \mathcal{L} }{ \partial b },
\end{align}$$

where α is the learning rate. Then repeat the process until the cost function stops decreasing.

The general **methodology** to build a neural network is to:

1. Define the neural network structure (# of input units, # of hidden units, etc).
2. Initialize the model's parameters
3. Loop:
    - Implement forward propagation (calculate the perceptron output),
    - Implement backward propagation (to get the required corrections for the parameters),
    - Update parameters.
4. Make predictions.

1.3 - Dataset

Load the [Kaggle dataset (https://www.kaggle.com/code/devzohaib/simple-linear-regression/notebook)](https://www.kaggle.com/code/devzohaib/simple-linear-regression/notebook), saved in a file `data/tvmarketing.csv` . It has two fields: TV marketing expenses (`TV`) and sales amount (`Sales`).

In [2]:

```
path = "data/tvmarketing.csv"

adv = pd.read_csv(path)
```

Print some part of the dataset.

In [3]:

```
adv.head()
```

Out[3]:

	TV	Sales
0	230.1	22.1
1	44.5	10.4
2	17.2	9.3
3	151.5	18.5
4	180.8	12.9

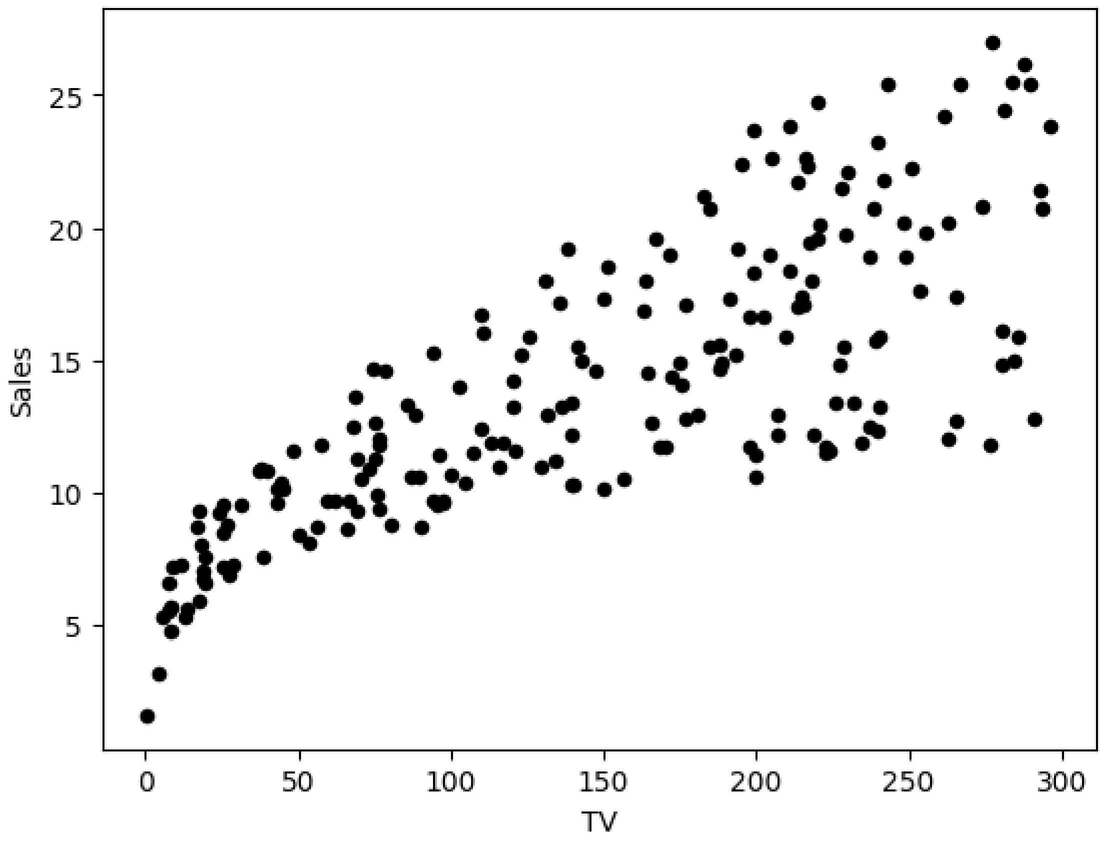
And plot it:

In [4]:

```
adv.plot(x='TV', y='Sales', kind='scatter', c='black')
```

Out[4]:

```
<AxesSubplot: xlabel='TV', ylabel='Sales'>
```



The fields `TV` and `Sales` have different units. Remember that in the week 2 assignment to make gradient descent algorithm efficient, you needed to normalize each of them: subtract the mean value of the array from each of the elements in the array and divide them by the standard deviation.

Column-wise normalization of the dataset can be done for all of the fields at once and is implemented in the following code:

In [5]:

```
adv_norm = (adv - np.mean(adv))/np.std(adv)
```

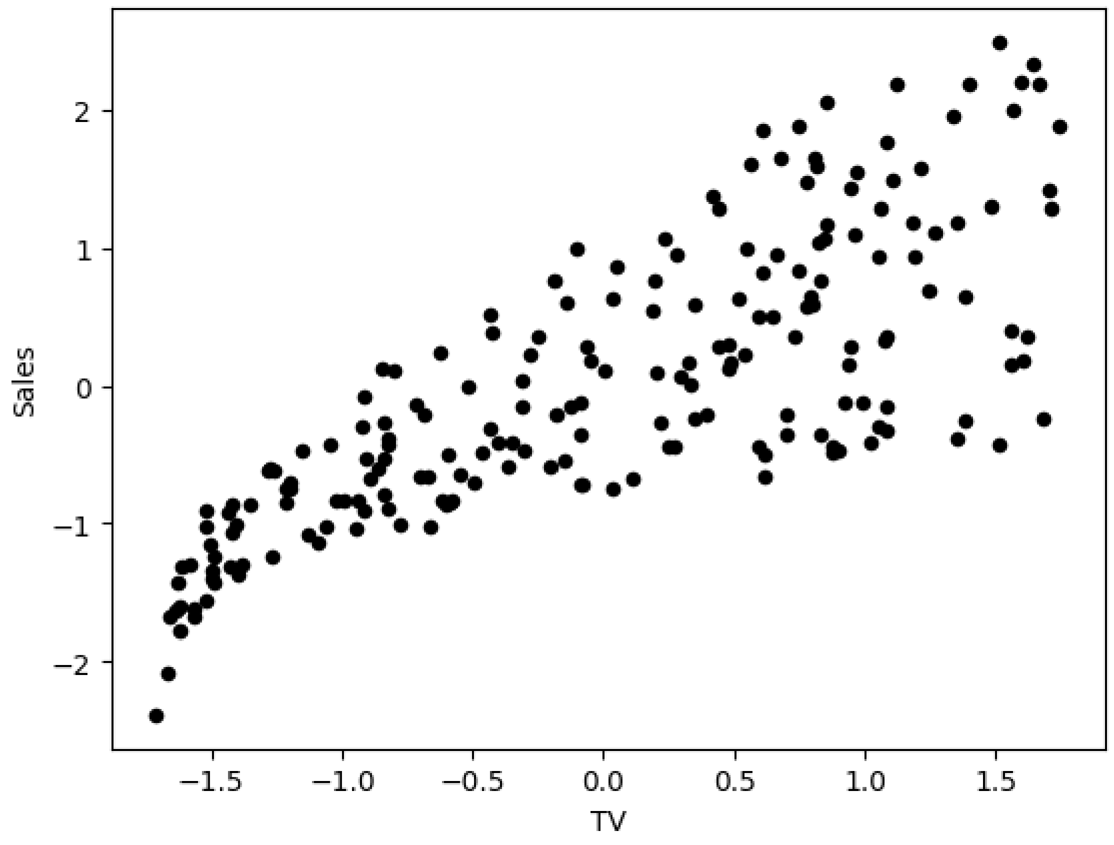
Plotting the data, you can see that it looks similar after normalization, but the values on the axes have changed:

In [6]:

```
adv_norm.plot(x='TV', y='Sales', kind='scatter', c='black')
```

Out[6]:

```
<AxesSubplot: xlabel='TV', ylabel='Sales'>
```



Save the fields into variables `X_norm` and `Y_norm` and reshape them to row vectors:

In [7]:

```
X_norm = adv_norm['TV']
Y_norm = adv_norm['Sales']

X_norm = np.array(X_norm).reshape((1, len(X_norm)))
Y_norm = np.array(Y_norm).reshape((1, len(Y_norm)))

print ('The shape of X_norm: ' + str(X_norm.shape))
print ('The shape of Y_norm: ' + str(Y_norm.shape))
print ('I have m = %d training examples!' % (X_norm.shape[1]))
```

```
The shape of X_norm: (1, 200)
The shape of Y_norm: (1, 200)
I have m = 200 training examples!
```

2 - Implementation of the Neural Network Model for Linear Regression

Setup the neural network in a way which will allow to extend this simple case of a model with a single perceptron and one input node to more complicated structures later.

2.1 - Defining the Neural Network Structure

Define two variables:

- n_x : the size of the input layer
- n_y : the size of the output layer

using shapes of arrays `X` and `Y` .

In [8]:

```
def layer_sizes(X, Y):
    """
    Arguments:
    X -- input dataset of shape (input size, number of examples)
    Y -- labels of shape (output size, number of examples)

    Returns:
    n_x -- the size of the input layer
    n_y -- the size of the output layer
    """
    n_x = X.shape[0]
    n_y = Y.shape[0]

    return (n_x, n_y)

(n_x, n_y) = layer_sizes(X_norm, Y_norm)
print("The size of the input layer is: n_x = " + str(n_x))
print("The size of the output layer is: n_y = " + str(n_y))
```

```
The size of the input layer is: n_x = 1
The size of the output layer is: n_y = 1
```

2.2 - Initialize the Model's Parameters

Implement the function `initialize_parameters()` , initializing the weights array of shape $(n_y \times n_x) = (1 \times 1)$ with random values and the bias vector of shape $(n_y \times 1) = (1 \times 1)$ with zeros.

In [9]:

```
def initialize_parameters(n_x, n_y):
    """
    Returns:
    params -- python dictionary containing your parameters:
                    W -- weight matrix of shape (n_y, n_x)
                    b -- bias value set as a vector of shape (n_y, 1)
    """

    W = np.random.randn(n_y, n_x) * 0.01
    b = np.zeros((n_y, 1))

    parameters = {"W": W,
                  "b": b}

    return parameters

parameters = initialize_parameters(n_x, n_y)
print("W = " + str(parameters["W"]))
print("b = " + str(parameters["b"]))
```

```
W = [[0.01788628]]
b = [[0.]]
```

2.3 - The Loop

Implement `forward_propagation()` following the equation (3) in the section 1.2:

$$Z = wX + b$$
$$\hat{Y} = Z,$$

In [10]:

```
def forward_propagation(X, parameters):
    """
    Argument:
    X -- input data of size (n_x, m)
    parameters -- python dictionary containing your parameters (output of initial

    Returns:
    Y_hat -- The output
    """
    W = parameters["W"]
    b = parameters["b"]

    # Forward Propagation to calculate Z.
    Z = np.matmul(W, X) + b
    Y_hat = Z

    return Y_hat

Y_hat = forward_propagation(X_norm, parameters)

print("Some elements of output vector Y_hat:", Y_hat[0, 0:5])
```

```
Some elements of output vector Y_hat: [ 0.01734705 -0.02141661 -0.02
711838  0.00093098  0.00705046]
```

Your weights were just initialized with some random values, so the model has not been trained yet.

Define a cost function (4) which will be used to train the model:

$$\mathcal{L}(w, b) = \frac{1}{2m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

In [11]:

```
def compute_cost(Y_hat, Y):
    """
    Computes the cost function as a sum of squares

    Arguments:
    Y_hat -- The output of the neural network of shape (n_y, number of examples)
    Y -- "true" labels vector of shape (n_y, number of examples)

    Returns:
    cost -- sum of squares scaled by 1/(2*number of examples)

    """
    # Number of examples.
    m = Y_hat.shape[1]

    # Compute the cost function.
    cost = np.sum((Y_hat - Y)**2)/(2*m)

    return cost

print("cost = " + str(compute_cost(Y_hat, Y_norm)))
```

```
cost = 0.48616887080159726
```

Calculate partial derivatives as shown in (5):

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)x^{(i)},$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right).$$

In [12]:

```
def backward_propagation(Y_hat, X, Y):
    """
    Implements the backward propagation, calculating gradients

    Arguments:
    Y_hat -- the output of the neural network of shape (n_y, number of examples)
    X -- input data of shape (n_x, number of examples)
    Y -- "true" labels vector of shape (n_y, number of examples)

    Returns:
    grads -- python dictionary containing gradients with respect to different par
    """
    m = X.shape[1]

    # Backward propagation: calculate partial derivatives denoted as dW, db for s
    dZ = Y_hat - Y
    dW = 1/m * np.dot(dZ, X.T)
    db = 1/m * np.sum(dZ, axis = 1, keepdims = True)

    grads = {"dW": dW,
             "db": db}

    return grads

grads = backward_propagation(Y_hat, X_norm, Y_norm)

print("dW = " + str(grads["dW"]))
print("db = " + str(grads["db"]))
```

```
dW = [[-0.76433814]]
db = [[5.19584376e-16]]
```

Update parameters as shown in (6):

$$w = w - \alpha \frac{\partial L}{\partial w},$$
$$b = b - \alpha \frac{\partial L}{\partial b}.$$

In [13]:

```python
def update_parameters(parameters, grads, learning_rate=1.2):
    """
    Updates parameters using the gradient descent update rule

    Arguments:
    parameters -- python dictionary containing parameters
    grads -- python dictionary containing gradients
    learning_rate -- learning rate parameter for gradient descent

    Returns:
    parameters -- python dictionary containing updated parameters
    """
    # Retrieve each parameter from the dictionary "parameters".
    W = parameters["W"]
    b = parameters["b"]

    # Retrieve each gradient from the dictionary "grads".
    dW = grads["dW"]
    db = grads["db"]

    # Update rule for each parameter.
    W = W - learning_rate * dW
    b = b - learning_rate * db

    parameters = {"W": W,
                  "b": b}

    return parameters

parameters_updated = update_parameters(parameters, grads)

print("W updated = " + str(parameters_updated["W"]))
print("b updated = " + str(parameters_updated["b"]))
```

```
W updated = [[0.93509205]]
b updated = [[-6.23501251e-16]]
```

2.4 - Integrate parts 2.1, 2.2 and 2.3 in nn_model() and make predictions

Build your neural network model in `nn_model()` .

In [14]:

```
def nn_model(X, Y, num_iterations=10, learning_rate=1.2, print_cost=False):
    """
    Arguments:
    X -- dataset of shape (n_x, number of examples)
    Y -- labels of shape (n_y, number of examples)
    num_iterations -- number of iterations in the loop
    learning_rate -- learning rate parameter for gradient descent
    print_cost -- if True, print the cost every iteration

    Returns:
    parameters -- parameters learnt by the model. They can then be used to make p
    """

    n_x = layer_sizes(X, Y)[0]
    n_y = layer_sizes(X, Y)[1]

    parameters = initialize_parameters(n_x, n_y)

    # Loop
    for i in range(0, num_iterations):

        # Forward propagation. Inputs: "X, parameters". Outputs: "Y_hat".
        Y_hat = forward_propagation(X, parameters)

        # Cost function. Inputs: "Y_hat, Y". Outputs: "cost".
        cost = compute_cost(Y_hat, Y)

        # Backpropagation. Inputs: "Y_hat, X, Y". Outputs: "grads".
        grads = backward_propagation(Y_hat, X, Y)

        # Gradient descent parameter update. Inputs: "parameters, grads, learning
        parameters = update_parameters(parameters, grads, learning_rate)

        # Print the cost every iteration.
        if print_cost:
            print ("Cost after iteration %i: %f" %(i, cost))

    return parameters
```

In [15]:

```
parameters_simple = nn_model(X_norm, Y_norm, num_iterations=30, learning_rate=1.2
print("W = " + str(parameters_simple["W"]))
print("b = " + str(parameters_simple["b"]))

W_simple = parameters["W"]
b_simple = parameters["b"]
```

```
Cost after iteration 0: 0.496595
Cost after iteration 1: 0.206164
Cost after iteration 2: 0.194547
Cost after iteration 3: 0.194082
Cost after iteration 4: 0.194063
Cost after iteration 5: 0.194063
Cost after iteration 6: 0.194062
Cost after iteration 7: 0.194062
Cost after iteration 8: 0.194062
Cost after iteration 9: 0.194062
Cost after iteration 10: 0.194062
Cost after iteration 11: 0.194062
Cost after iteration 12: 0.194062
Cost after iteration 13: 0.194062
Cost after iteration 14: 0.194062
Cost after iteration 15: 0.194062
Cost after iteration 16: 0.194062
Cost after iteration 17: 0.194062
Cost after iteration 18: 0.194062
Cost after iteration 19: 0.194062
Cost after iteration 20: 0.194062
Cost after iteration 21: 0.194062
Cost after iteration 22: 0.194062
Cost after iteration 23: 0.194062
Cost after iteration 24: 0.194062
Cost after iteration 25: 0.194062
Cost after iteration 26: 0.194062
Cost after iteration 27: 0.194062
Cost after iteration 28: 0.194062
Cost after iteration 29: 0.194062
W = [[0.78222442]]
b = [[-6.07514039e-16]]
```

You can see that after a few iterations the cost function does not change anymore (the model converges).

Note: This is a very simple model. In reality the models do not converge that quickly.

The final model parameters can be used for making predictions, but don't forget about normalization and denormalization.

In [16]:

```
def predict(X, Y, parameters, X_pred):

    # Retrieve each parameter from the dictionary "parameters".
    W = parameters["W"]
    b = parameters["b"]

    # Use the same mean and standard deviation of the original training array X.
    if isinstance(X, pd.Series):
        X_mean = np.mean(X)
        X_std = np.std(X)
        X_pred_norm = ((X_pred - X_mean)/X_std).reshape((1, len(X_pred)))
    else:
        X_mean = np.array(np.mean(X)).reshape((len(X.axes[1]),1))
        X_std = np.array(np.std(X)).reshape((len(X.axes[1]),1))
        X_pred_norm = ((X_pred - X_mean)/X_std)
    # Make predictions.
    Y_pred_norm = np.matmul(W, X_pred_norm) + b
    # Use the same mean and standard deviation of the original training array Y.
    Y_pred = Y_pred_norm * np.std(Y) + np.mean(Y)

    return Y_pred[0]

X_pred = np.array([50, 120, 280])
Y_pred = predict(adv["TV"], adv["Sales"], parameters_simple, X_pred)
print(f"TV marketing expenses:\n{X_pred}")
print(f"Predictions of sales:\n{Y_pred}")
```

```
TV marketing expenses:
[ 50 120 280]
Predictions of sales:
[ 9.40942557 12.7369904  20.34285287]
```

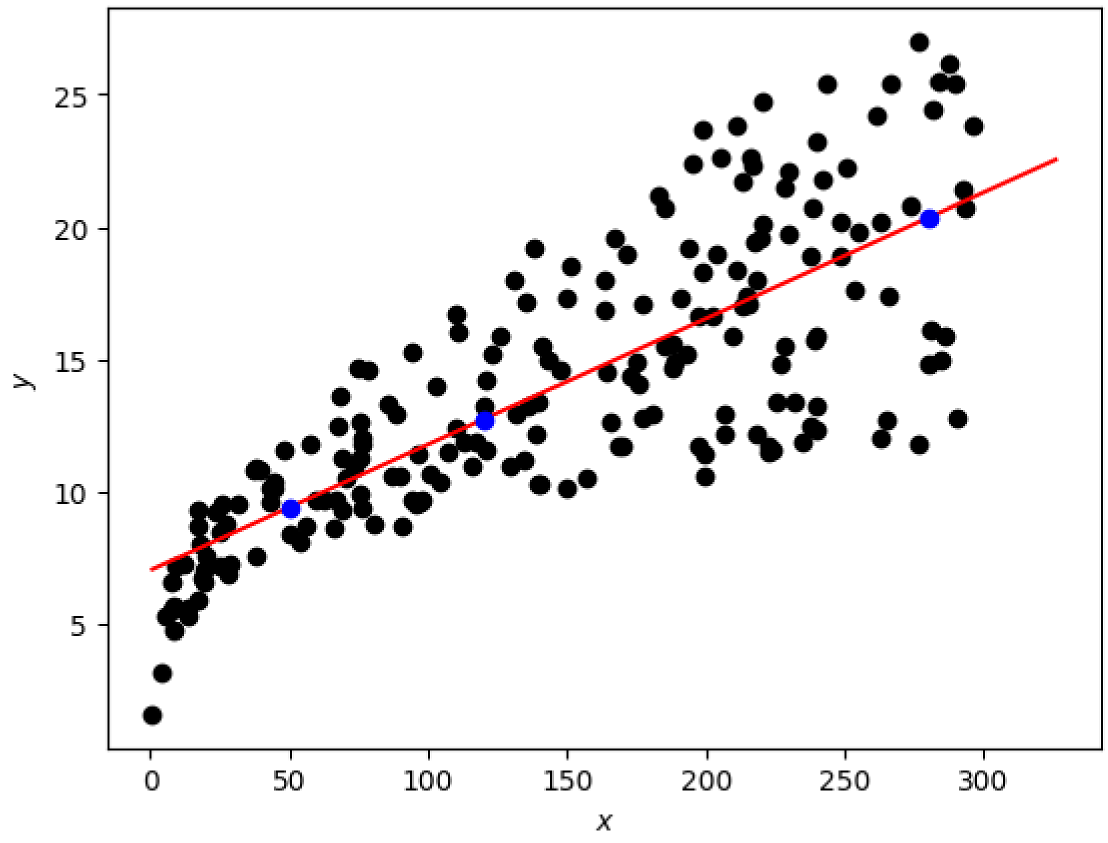
Let's plot the linear regression line and some predictions. The regression line is red and the predicted points are blue.

In [17]:

```
fig, ax = plt.subplots()
plt.scatter(adv["TV"], adv["Sales"], color="black")

plt.xlabel("$x$")
plt.ylabel("$y$")

X_line = np.arange(np.min(adv["TV"]),np.max(adv["TV"])*1.1, 0.1)
Y_line = predict(adv["TV"], adv["Sales"], parameters_simple, X_line)
ax.plot(X_line, Y_line, "r")
ax.plot(X_pred, Y_pred, "bo")
plt.plot()
plt.show()
```



Now let's increase the number of the input nodes to build a multiple linear regression model.

3 - Multiple Linear Regression

3.1 - Multipe Linear Regression Model

You can write a multiple linear regression model with two independent variables x_1, x_2 as

$$\hat{y} = w_1x_1 + w_2x_2 + b = Wx + b,$$

where Wx is the dot product of the input vector $x = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$ and the parameters vector $W = \begin{bmatrix} w_1 & w_2 \end{bmatrix}$, scalar parameter b is the intercept. The goal of the training process is to find the "best" parameters w_1, w_2 and b such that the differences between original values y_i and predicted values $\hat{y}_i$ are minimum for the given training examples.

3.2 - Neural Network Model with a Single Perceptron and Two Input Nodes

To describe the multiple regression problem, you can still use a model with one perceptron, but this time you need two input nodes, as shown in the following scheme:

The perceptron output calculation for every training example $x^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} \end{bmatrix}$ can be written with dot product:

$$z^{(i)} = w_1x_1^{(i)} + w_2x_2^{(i)} + b = Wx^{(i)} + b,$$

where weights are in the vector $W = \begin{bmatrix} w_1 & w_2 \end{bmatrix}$ and bias b is a scalar. The output layer will have the same single node $\hat{y} = z$.

Organise all training examples in a matrix X of a shape $(2 \times m)$, putting $x_1^{(i)}$ and $x_2^{(i)}$ into columns. Then matrix multiplication of W (1×2) and X $(2 \times m)$ will give a $(1 \times m)$ vector

$$WX = \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \dots & x_2^{(m)} \end{bmatrix} = \begin{bmatrix} w_1x_1^{(1)} + w_2x_2^{(1)} & w_1x_1^{(2)} + w_2x_2^{(2)} & \dots & w_1x_1^{(m)} + w_2x_2^{(m)} \end{bmatrix}.$$

And the model can be written as

$$\begin{aligned} Z &= WX + b, \\ \hat{Y} &= Z, \end{aligned}$$

where b is broadcasted to the vector of size $(1 \times m)$. These are the calculations to perform in the forward propagation step. Cost function will remain the same (see equation (4) in the section 1.2):

$$\mathcal{L}(w, b) = \frac{1}{2m}\sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right)^2.$$

To implement the gradient descent algorithm, you can calculate cost function partial derivatives as:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_1} &= \frac{1}{m}\sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right)x_1^{(i)}, \\ \frac{\partial \mathcal{L}}{\partial w_2} &= \frac{1}{m}\sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right)x_2^{(i)}, \\ \frac{\partial \mathcal{L}}{\partial b} &= \frac{1}{m}\sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right). \end{aligned}$$

After performing the forward propagation as shown in (9), the variable $\hat{Y}$ will contain the predictions in the array of size $(1 \times m)$. The original values $y^{(i)}$ will be kept in the array Y of the same size. Thus, $\left(\hat{Y} - Y\right)$ will be a $(1 \times m)$ array containing differences $\left(\hat{y}^{(i)} - y^{(i)}\right)$. Matrix X of size $(2 \times m)$ has all $x_1^{(i)}$ values in the first row and $x_2^{(i)}$ in the second row. Thus, the sums in the first two equations of (10) can be calculated as matrix multiplication of $\left(\hat{Y} - Y\right)$ of a shape $(1 \times m)$ and X^T of a shape $(m \times 2)$, resulting in the (1×2) array:

$$\frac{\partial \mathcal{L}}{\partial W} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1} & \frac{\partial \mathcal{L}}{\partial w_2} \end{bmatrix} = \frac{1}{m}\left(\hat{Y} - Y\right)X^T.$$

Similarly for $\frac{\partial \mathcal{L}}{\partial b}$:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m}\left(\hat{Y} - Y\right)\mathbf{1}.$$

where $\mathbf{1}$ is just a $(m \times 1)$ vector of ones.

See how linear algebra and calculus work together to make calculations so nice and tidy! You can now update the parameters using matrix form of W:

$$W = W - \alpha \frac{\partial \mathcal{L}}{\partial W},$$

3.3 - Dataset

Let's build a linear regression model for a Kaggle dataset House Prices (https://www.kaggle.com/c/house-prices-advanced-regression-techniques), saved in a file `data/house_prices_train.csv` . You will use two fields - ground living area (`GrLivArea` , square feet) and rates of the overall quality of material and finish (`OverallQual` , 1-10) to predict sales price (`SalePrice` , dollars).

To open the dataset you can use `pandas` function `read_csv` :

In [18]:

```
df = pd.read_csv('data/house_prices_train.csv')
```

Select the required fields and save them in the variables `X_multi` , `Y_multi` :

In [19]:

```
X_multi = df[['GrLivArea', 'OverallQual']]
Y_multi = df['SalePrice']
```

Preview the data:

In [20]:

```
display(X_multi)
display(Y_multi)
```

	GrLivArea	OverallQual
0	1710	7
1	1262	6
2	1786	7
3	1717	7
4	2198	8
...	...	...
1455	1647	6
1456	2073	6
1457	2340	7
1458	1078	5
1459	1256	5

1460 rows × 2 columns

```
0       208500
1       181500
2       223500
3       140000
4       250000
         ...
1455    175000
1456    210000
1457    266500
1458    142125
1459    147500
Name: SalePrice, Length: 1460, dtype: int64
```

Normalize the data:

In [21]:

```
X_multi_norm = (X_multi - np.mean(X_multi))/np.std(X_multi)
Y_multi_norm = (Y_multi - np.mean(Y_multi))/np.std(Y_multi)
```

Convert results to the `NumPy` arrays, transpose `X_multi_norm` to get an array of a shape $(2 \times m)$ and reshape `Y_multi_norm` to bring it to the shape $(1 \times m)$:

In [22]:

```
X_multi_norm = np.array(X_multi_norm).T
Y_multi_norm = np.array(Y_multi_norm).reshape((1, len(Y_multi_norm)))

print ('The shape of X: ' + str(X_multi_norm.shape))
print ('The shape of Y: ' + str(Y_multi_norm.shape))
print ('I have m = %d training examples!' % (X_multi_norm.shape[1]))
```

```
The shape of X: (2, 1460)
The shape of Y: (1, 1460)
I have m = 1460 training examples!
```

3.4 - Performance of the Neural Network Model for Multiple Linear Regression

Now... you do not need to change anything in your neural network implementation! Go through the code in section 2 and see that if you pass new datasets `X_multi_norm` and `Y_multi_norm`, the input layer size n_x will get equal to 2 and the rest of the implementation will remain exactly the same, even the backward propagation!

Train the model for 100 iterations:

In [23]:

```
parameters_multi = nn_model(X_multi_norm, Y_multi_norm, num_iterations=100, print_

print("W = " + str(parameters_multi["W"]))
print("b = " + str(parameters_multi["b"]))

W_multi = parameters_multi["W"]
b_multi = parameters_multi["b"]
```

```
Cost after iteration 0: 0.514220
Cost after iteration 1: 0.448627
Cost after iteration 2: 0.396224
Cost after iteration 3: 0.353227
Cost after iteration 4: 0.317639
Cost after iteration 5: 0.288102
Cost after iteration 6: 0.263566
Cost after iteration 7: 0.243179
Cost after iteration 8: 0.226237
Cost after iteration 9: 0.212158
Cost after iteration 10: 0.200457
Cost after iteration 11: 0.190734
Cost after iteration 12: 0.182654
Cost after iteration 13: 0.175939
Cost after iteration 14: 0.170359
Cost after iteration 15: 0.165721
Cost after iteration 16: 0.161867
Cost after iteration 17: 0.158665
Cost after iteration 18: 0.156003
Cost after iteration 19: 0.153792
Cost after iteration 20: 0.151953
Cost after iteration 21: 0.150426
Cost after iteration 22: 0.149157
Cost after iteration 23: 0.148102
Cost after iteration 24: 0.147225
Cost after iteration 25: 0.146496
Cost after iteration 26: 0.145891
Cost after iteration 27: 0.145388
Cost after iteration 28: 0.144970
Cost after iteration 29: 0.144622
Cost after iteration 30: 0.144334
Cost after iteration 31: 0.144094
Cost after iteration 32: 0.143894
Cost after iteration 33: 0.143728
Cost after iteration 34: 0.143591
Cost after iteration 35: 0.143476
Cost after iteration 36: 0.143381
Cost after iteration 37: 0.143302
Cost after iteration 38: 0.143236
Cost after iteration 39: 0.143182
Cost after iteration 40: 0.143136
Cost after iteration 41: 0.143099
Cost after iteration 42: 0.143067
Cost after iteration 43: 0.143041
Cost after iteration 44: 0.143020
Cost after iteration 45: 0.143002
Cost after iteration 46: 0.142987
Cost after iteration 47: 0.142974
Cost after iteration 48: 0.142964
Cost after iteration 49: 0.142956
Cost after iteration 50: 0.142948
Cost after iteration 51: 0.142943
Cost after iteration 52: 0.142938
Cost after iteration 53: 0.142934
Cost after iteration 54: 0.142930
Cost after iteration 55: 0.142927
Cost after iteration 56: 0.142925
Cost after iteration 57: 0.142923
Cost after iteration 58: 0.142921
Cost after iteration 59: 0.142920
Cost after iteration 60: 0.142919
Cost after iteration 61: 0.142918
Cost after iteration 62: 0.142917
Cost after iteration 63: 0.142917
Cost after iteration 64: 0.142916
Cost after iteration 65: 0.142916
Cost after iteration 66: 0.142915
Cost after iteration 67: 0.142915
Cost after iteration 68: 0.142915
Cost after iteration 69: 0.142914
Cost after iteration 70: 0.142914
Cost after iteration 71: 0.142914
Cost after iteration 72: 0.142914
Cost after iteration 73: 0.142914
Cost after iteration 74: 0.142914
Cost after iteration 75: 0.142914
Cost after iteration 76: 0.142914
Cost after iteration 77: 0.142914
Cost after iteration 78: 0.142914
Cost after iteration 79: 0.142914
Cost after iteration 80: 0.142914
Cost after iteration 81: 0.142914
Cost after iteration 82: 0.142913
Cost after iteration 83: 0.142913
Cost after iteration 84: 0.142913
Cost after iteration 85: 0.142913
Cost after iteration 86: 0.142913
Cost after iteration 87: 0.142913
Cost after iteration 88: 0.142913
Cost after iteration 89: 0.142913
Cost after iteration 90: 0.142913
Cost after iteration 91: 0.142913
Cost after iteration 92: 0.142913
Cost after iteration 93: 0.142913
Cost after iteration 94: 0.142913
Cost after iteration 95: 0.142913
Cost after iteration 96: 0.142913
Cost after iteration 97: 0.142913
Cost after iteration 98: 0.142913
Cost after iteration 99: 0.142913
W = [[0.3694604  0.57181575]]
b = [[1.02931362e-16]]
```

Now you are ready to make predictions:

In [24]:

```
X_pred_multi = np.array([[1710, 7], [1200, 6], [2200, 8]]).T
Y_pred_multi = predict(X_multi, Y_multi, parameters_multi, X_pred_multi)

print(f"Ground living area, square feet:\n{X_pred_multi[0]}")
print(f"Rates of the overall quality of material and finish, 1-10:\n{X_pred_multi
print(f"Predictions of sales price, $:\n{np.round(Y_pred_multi)}")
```

```
Ground living area, square feet:
[1710 1200 2200]
Rates of the overall quality of material and finish, 1-10:
[7 6 8]
Predictions of sales price, $:
[221371. 160039. 281587.]
```

Congrats on finishing this lab!

In [ ]:

```

```