C1W3_Assignment_for_LinearAlgebra[ML] Submitted by_Ishraq

December 30, 2024

1 Linear Transformations and Neural Networks

2 Outline

Welcome to the third assignment of the Linear Algebra course!

This assignment explores the foundational concepts of linear transformations and neural networks in two distinct parts. In the first section, we delve into linear transformations by creating functions to generate matrices for stretching, shearing, and rotation operations. The second part shifts the focus to neural networks, specifically implementing forward propagation in a simple architecture with two inputs and one perceptron. By dissecting these fundamental components, this assignment aims to provide a clear understanding of the role of linear algebra in both vector transformations and neural network computations.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import utils
```

[ ]:
```
import w3_unittest
```

## 1 - Introduction

### 1.1 - Transformations

A **transformation** is a function from one vector space to another that respects the underlying (linear) structure of each vector space. Referring to a specific transformation, you can use a symbol, such as T. Specifying the spaces containing the input and output vectors, e.g. $\mathbb{R}^2$ and $\mathbb{R}^3$, you can write $T: \mathbb{R}^2 \rightarrow \mathbb{R}^3$. Transforming vector $v \in \mathbb{R}^2$ into the vector $w \in \mathbb{R}^3$ by the transformation T, you can use the notation $T(v) = w$ and read it as "*T of v equals to w*" or "*vector w is an **image** of vector v with the transformation T*".

The following Python function corresponds to the transformation $T: \mathbb{R}^2 \rightarrow \mathbb{R}^3$ with the following symbolic formula:

$$T\left(\begin{bmatrix} v_1 \\ v_2 \end{bmatrix}\right) = \begin{bmatrix} 3v_1 \\ 0 \\ -2v_2 \end{bmatrix} \tag{1}$$

[ ]:
```
def T(v):
    w = np.zeros((3,1))
    w[0,0] = 3*v[0,0]
    w[2,0] = -2*v[1,0]

    return w

v = np.array([[3], [5]])
w = T(v)

print("Original vector:\n", v, "\n\n Result of the transformation:\n", w)
```

### 1.2 - Linear Transformations

A transformation T is said to be **linear** if the following two properties are true for any scalar k, and any input vectors u and v:

1. $T(kv) = kT(v)$,
2. $T(u + v) = T(u) + T(v)$.

In the example above T is a linear transformation:

$$T(kv) = T\left(\begin{bmatrix} kv_1 \\ kv_2 \end{bmatrix}\right) = \begin{bmatrix} 3kv_1 \\ 0 \\ -2kv_2 \end{bmatrix} = k\begin{bmatrix} 3v_1 \\ 0 \\ -2v_2 \end{bmatrix} = kT(v), \tag{2}$$

$$T(u+v) = T\left(\begin{bmatrix} u_1 + v_1 \\ u_2 + v_2 \end{bmatrix}\right) = \begin{bmatrix} 3(u_1+v_1) \\ 0 \\ -2(u_2+v_2) \end{bmatrix} = \begin{bmatrix} 3u_1 \\ 0 \\ -2u_2 \end{bmatrix} + \begin{bmatrix} 3v_1 \\ 0 \\ -2v_2 \end{bmatrix} = T(u) + T(v). \tag{3}$$

You can change the values of k or vectors u and v in the cell below, to check that this is true for some specific values.

```
[ ]: u = np.array([[1], [-2]])
     v = np.array([[2], [4]])

     k = 7

     print("T(k*v):\n", T(k*v), "\n k*T(v):\n", k*T(v), "\n\n")
     print("T(u+v):\n", T(u+v), "\n\n T(u)+T(v):\n", T(u)+T(v))
```

### 1.3 - Transformations Defined as a Matrix Multiplication

Let $L : \mathbb{R}^m \rightarrow \mathbb{R}^n$ be defined by a matrix A, where $L(v) = Av$, multiplication of the matrix A $(n \times m)$ and vector v $(m \times 1)$ resulting in the vector w $(n \times 1)$.

Now try to guess, what should be the elements of matrix A, corresponding to the transformation $L : \mathbb{R}^2 \rightarrow \mathbb{R}^3$:

$$L\left(\begin{bmatrix} v_1 \\ v_2 \end{bmatrix}\right) = \begin{bmatrix} 3v_1 \\ 0 \\ -2v_2 \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \\ ? & ? \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \tag{4}$$

To do that, write the transformation L as Av and then perform matrix multiplication:

$$L\left(\begin{bmatrix} v_1 \\ v_2 \end{bmatrix}\right) = A\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} a_{1,1}v_1 + a_{1,2}v_2 \\ a_{2,1}v_1 + a_{2,2}v_2 \\ a_{3,1}v_1 + a_{3,2}v_2 \end{bmatrix} = \begin{bmatrix} 3v_1 \\ 0 \\ -2v_2 \end{bmatrix} \tag{5}$$

Can you see now what should be the values of the elements $a_{i,j}$ of matrix A to make the equalities (5) correct? Find out the answer in the following code cell:

```
[ ]: def L(v):
         A = np.array([[3,0], [0,0], [0,-2]])
         print("Transformation matrix:\n", A, "\n")
         w = A @ v

         return w

     v = np.array([[3], [5]])
     w = L(v)
```

```
print("Original vector:\n", v, "\n\n Result of the transformation:\n", w)
```

## 2 - Standard Transformations in a Plane

As discussed above in section 1, a linear transformation $L : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ can be represented as a multiplication of a 2×2 matrix and a coordinate vector $v \in \mathbb{R}^2$. Note that so far you have been using some random vector $v \in \mathbb{R}^2$. (e.g. $v = \begin{bmatrix}3\\5\end{bmatrix}$). To have a better intuition of what the transformation is really doing in the $\mathbb{R}^2$ space, it is wise to choose vector v in a less random way.

A good choice would be vectors of a standard basis $e_1 = \begin{bmatrix}1\\0\end{bmatrix}$ and $e_2 = \begin{bmatrix}0\\1\end{bmatrix}$. Let's apply linear transformation L to each of the vectors e_1 and e_2: $L(e_1) = Ae_1$ and $L(e_2) = Ae_2$. If you put vectors $\{e_1, e_2\}$ into columns of a matrix and perform matrix multiplication

$$A\begin{bmatrix}e_1 & e_2\end{bmatrix} = \begin{bmatrix}Ae_1 & Ae_2\end{bmatrix} = \begin{bmatrix}L(e_1) & L(e_2)\end{bmatrix}, \tag{3}$$

you can note that $\begin{bmatrix}e_1 & e_2\end{bmatrix} = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$ (identity matrix). Thus, $A\begin{bmatrix}e_1 & e_2\end{bmatrix} = AI = A$, and

$$A = \begin{bmatrix}L(e_1) & L(e_2)\end{bmatrix}. \tag{4}$$

This is a matrix with the columns that are the images of the vectors of the standard basis.

This choice of vectors $\{e_1, e_2\}$ provides opportinuty for the visual representation of the linear transformation L (you will see the examples below).

You will work with an image where you will see every transformation working in practice. Since an image is just a set of points in the plane, you can work with it as a set of vectors and you can stretch them, rotate them and much more!

[2]:
```
img = np.loadtxt('data/image.txt')
print('Shape: ',img.shape)
print(img)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-a3b44e7177f6> in <module>
----> 1 img = np.loadtxt('data/image.txt')
      2 print('Shape: ',img.shape)
      3 print(img)

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

The image is just a 2×329076 matrix where each column is a vector in the plane. So you can get all x coordinates by calling `img[0]` and all y coordinates by calling `img[1]`. Now let's plot the image!

[3]:
```
plt.scatter(img[0], img[1], s = 0.001, color = 'black')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-7e8f9e99c021> in <module>
----> 1 plt.scatter(img[0], img[1], s = 0.001, color = 'black')

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

### 2.1 - Horizontal Scaling (Dilation)

Horizontal scaling (factor 2 in this example) can be defined considering transformation of a vector $e_1 = \begin{bmatrix}1\\0\end{bmatrix}$ into a vector $\begin{bmatrix}2\\0\end{bmatrix}$ and leaving vector $e_2 = \begin{bmatrix}0\\1\end{bmatrix}$ without any changes. The following function `T_hscaling()` corresponds to the horizontal scaling (factor 2) of a vector. The second function `transform_vectors()` applies defined transformation to a set of vectors (here two vectors).

[4]:
```
def T_hscaling(v):
    A = np.array([[2,0], [0,1]])
    w = A @ v

    return w


def transform_vectors(T, v1, v2):
    V = np.hstack((v1, v2))
    W = T(V)

    return W

e1 = np.array([[1], [0]])
e2 = np.array([[0], [1]])

transformation_result_hscaling = transform_vectors(T_hscaling, e1, e2)

print("Original vectors:\n e1= \n", e1, "\n e2=\n", e2,
      "\n\n Result of the transformation (matrix form):\n",␣
 ↪transformation_result_hscaling)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-5de3f1d50524> in <module>
     12     return W
     13
---> 14 e1 = np.array([[1], [0]])
     15 e2 = np.array([[0], [1]])
     16
```

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

You can get a visual understanding of the transformation, producing a plot which displays input vectors, and their transformations. We have written this function for you so, so do not need to worry! Just run the next cell.

[ ]:
```
utils.plot_transformation(T_hscaling,e1,e2)
```

Let's visualize how it acts in our leaf image.

[ ]:
```
plt.scatter(img[0], img[1], s = 0.001, color = 'black')
plt.scatter(T_hscaling(img)[0], T_hscaling(img)[1], s = 0.001, color = 'grey')
```

Note that the transformed image (grey) is stretched!

### 2.2 - Example 2: Reflection about y-axis (the vertical axis)

Function `T_reflection_yaxis()` defined below corresponds to the reflection about y-axis:

[ ]:
```
def T_reflection_yaxis(v):
    A = np.array([[-1,0], [0,1]])
    w = A @ v

    return w

e1 = np.array([[1], [0]])
e2 = np.array([[0], [1]])

transformation_result_reflection_yaxis = transform_vectors(T_reflection_yaxis,␣
↪e1, e2)

print("Original vectors:\n e1= \n", e1,"\n e2=\n", e2,
      "\n\n Result of the transformation (matrix form):\n",␣
↪transformation_result_reflection_yaxis)
```

You can visualize this transformation:

[ ]:
```
utils.plot_transformation(T_reflection_yaxis, e1, e2)
```

[ ]:
```
plt.scatter(img[0], img[1], s = 0.001, color = 'black')
plt.scatter(T_reflection_yaxis(img)[0], T_reflection_yaxis(img)[1], s = 0.001,␣
↪color = 'grey')
```

### 2.3 Stretching by a scalar

The next linear transformation you will work is a stretching by a non-zero scalar. In other words, fixing a scalar $a \neq 0$, the linear transformation in the plane is such that:

$$T(v) = a \cdot v$$

If $v = (x, y)$, then $T(v) = T((x, y)) = (ax, ay)$.

### Exercise 1

In the following exercise, you will write a function that inputs a non-zero scalar a and a vector v and stretches v in a factor of a in the plane.

Hint: To find the matrix, you may proceed as you just saw above: try checking the result in the *special* vectors $e_1 = (1, 0)$ and $e_2 = (0, 1)$!

[ ]:
```
# GRADED FUNCTION: T_stretch

def T_stretch(a, v):
    """
    Performs a 2D stretching transformation on a vector v using a stretching␣
 ↪factor a.

    Args:
        a (float): The stretching factor.
        v (numpy.array): The vector (or vectors) to be stretched.

    Returns:
        numpy.array: The stretched vector.
    """

    ### START CODE HERE ###
    # Define the transformation matrix
    T = np.array([[a,0], [0,a]])

    # Compute the transformation
    w = T @ v
    ### END CODE HERE ###

    return w
```

[ ]:
```
w3_unittest.test_T_stretch(T_stretch)
```

[ ]:
```
plt.scatter(img[0], img[1], s = 0.001, color = 'black')
plt.scatter(T_stretch(2,img)[0], T_stretch(2,img)[1], s = 0.001, color = 'grey')
```

[ ]:
```
utils.plot_transformation(lambda v: T_stretch(2, v), e1,e2)
```

### 2.4 Horizontal shear transformation

The horizontal shear transformation with parameter m is a transformation that maps a point (x, y) in the plane to $(x + my, y)$. In other words,

$$T((x,y)) = (x + my, y)$$

### Exercise 2

You will implement a function `T_hshear` that inputs a scalar m and a vector v, and performs the transformation mentioned above.

Hint: To find the associated matrix you may want to understand what happens with the vectors $e_1 = (1, 0)$ and $e_2 = (0, 1)$.

```
[ ]: # GRADED FUNCTION: T_hshear

def T_hshear(m, v):
    """
    Performs a 2D horizontal shearing transformation on an array v using a␣
 ↪shearing factor m.

    Args:
        m (float): The shearing factor.
        v (np.array): The array to be sheared.

    Returns:
        np.array: The sheared array.
    """

    ### START CODE HERE ###
    # Define the transformation matrix
    T = np.array([[1,m], [0,1]])

    # Compute the transformation
    w = T @ v

    ### END CODE HERE ###

    return w
```

```
[ ]: w3_unittest.test_T_hshear(T_hshear)
```

```
[ ]: plt.scatter(img[0], img[1], s = 0.001, color = 'black')
     plt.scatter(T_hshear(2,img)[0], T_hshear(2,img)[1], s = 0.001, color = 'grey')
```

```
[ ]: utils.plot_transformation(lambda v: T_hshear(2, v), e1,e2)
```

### 2.5 Rotation

To rotate a vector in the plane by an angle of θ (radians), the matrix related to this transformation is given by:

$$M = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

### Exercise 3

You will implement the function `T_rotation` that inputs an angle in radians, θ, a vector v, and performs a rotation by θ counterclockwise.

[ ]:
```
# GRADED FUNCTION: T_rotation
def T_rotation(theta, v):
    """
    Performs a 2D rotation transformation on an array v using a rotation angle␣
 ↪theta.

    Args:
        theta (float): The rotation angle in radians.
        v (np.array): The array to be rotated.

    Returns:
        np.array: The rotated array.
    """

    ### START CODE HERE ###
    # Define the transformation matrix
    T = np.array([[np.cos(theta),-np.sin(theta)], [np.sin(theta),np.
 ↪cos(theta)]])

    # Compute the transformation
    w = T @ v

    ### END CODE HERE ###

    return w
```

[ ]:
```
w3_unittest.test_T_rotation(T_rotation)
```

[ ]:
```
plt.scatter(img[0], img[1], s = 0.001, color = 'black')
plt.scatter(T_rotation(np.pi,img)[0], T_rotation(np.pi,img)[1], s = 0.001,␣
 ↪color = 'grey')
```

[ ]:
```
utils.plot_transformation(lambda v: T_rotation(np.pi, v), e1,e2)
```

### Exercise 4

In the last exercise of this section, you will implement a function that rotates by an angle of θ and stretches by a factor of a. Remember that if T_{stretch} is the stretching transformation and T_{rotation} is the rotation then

$$T_{\text{rotation and stretch}}(v) = \left(T_{\text{rotation}} \circ T_{\text{stretch}}\right)(v) = T_{\text{rotation}}\left(T_{\text{stretch}}\left(v\right)\right).$$

So to perform both transformations, you must multiply the corresponding matrices!

```
[ ]: def T_rotation_and_stretch(theta, a, v):
         """
         Performs a combined 2D rotation and stretching transformation on an array v␣
      ↪using a rotation angle theta and a stretching factor a.

         Args:
             theta (float): The rotation angle in radians.
             a (float): The stretching factor.
             v (np.array): The array to be transformed.

         Returns:
             np.array: The transformed array.
         """
         ### START CODE HERE ###

         rotation_T = np.array([
             [np.cos(theta), -np.sin(theta)],
             [np.sin(theta), np.cos(theta)]
         ])
         stretch_T = np.array([
             [a, 0],
             [0, a]
         ])

         w = rotation_T @ (stretch_T @ v)

         ### END CODE HERE ###

         return w
```

```
[ ]: w3_unittest.test_T_rotation_and_stretch(T_rotation_and_stretch)
```

```
[ ]: plt.scatter(img[0], img[1], s = 0.001, color = 'black')
     plt.scatter(T_rotation_and_stretch(np.pi,2,img)[0], T_rotation_and_stretch(np.
      ↪pi,2,img)[1], s = 0.001, color = 'grey')
```

```
[ ]: utils.plot_transformation(lambda v: T_rotation_and_stretch(np.pi, 2, v), e1,e2)
```

## 3 - Neural Networks

In this part of the assignment you will:

- Implement a neural network with a single perceptron and two input nodes for linear regression
- Implement forward propagation using matrix multiplication

Note: Backward propagation with the parameters update requires understanding of Calculus. It is discussed in details in the Course “Calculus” (Course 2 in the Specialization “Mathematics for Machine Learning”). In this assignment backward propagation and parameters update functions are hidden.

### 3.1 - Linear Regression

Linear regression is a linear approach for modelling the relationship between a scalar response (**dependent variable**) and one or more explanatory variables (**independent variables**). You will work with a linear regression with 2 independent variables.

Linear regression model with two independent variables x_1, x_2 can be written as

$$\hat{y} = w_1x_1 + w_2x_2 + b = Wx + b, \tag{6}$$

where Wx is the dot product of the input vector $x = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$ and parameters vector $W = \begin{bmatrix} w_1 & w_2 \end{bmatrix}$, scalar parameter b is the intercept.

The goal is the same - find the “best” parameters w_1, w_2 and b such the differences between original values y_i and predicted values $\hat{y}_i$ are minimum.

You can use a neural network model to do that. Matrix multiplication will be in the core of the model!

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

Again, you will use only one perceptron, but with two input nodes shown in the following scheme:

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

$$z = w_1x_1 + w_2x_2 + b = Wx + b$$

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 and x_2 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

$$Z = WX + b, \tag{1}$$
$$\hat{Y} = Z, \tag{8}$$

where b is broadcasted to the vector of a size $(1 \times m)$. These are the calculations to perform in the forward propagation step.

Now, you can compare the resulting vector of the predictions $\hat{Y}$ $(1 \times m)$ with the original vector of data Y. This can be done with the so called **cost function** that measures how close your vector of predictions is to the training data. It evaluates how well the parameters w and b work to solve the problem. There are many different cost functions available depending on the nature of your problem. For your simple neural network you can calculate it as:

$$\mathcal{L}\left(w, b\right) = \frac{1}{2m}\sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{5}$$

The aim is to minimize the cost function during the training, which will minimize the differences between original values y_i and predicted values $\hat{y}_i$ (division by $2m$ is taken just for scaling purposes).

When your weights were just initialized with some random values, and no training was done yet, you can't expect good results.

The next step is to adjust the weights and bias, in order to minimize the cost function. This process is called **backward propagation** and is done iteratively: you update the parameters with a small change and repeat the process.

Note: Backward propagation is not covered in this Course - it will be discussed in the next Course of this Specialization.

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.

### 3.3 Parameters of the Neural Network

The neural network you will be working with has 3 parameters. Two weights and one bias, you will start initalizing these parameters as some random numbers, so the algorithm can start at some point. The parameters will be stored in a dictionary.

```
[ ]: parameters = utils.initialize_parameters(2)
     print(parameters)
```

### 3.4 Forward propagation

### Exercise 5

Implement `forward_propagation()`.

Instructions: - Look at the mathematical representation of your model:

$$Z = WX + b \tag{2}$$
$$\hat{Y} = Z, \tag{3}$$

- The steps you have to implement are: 1. Retrieve each parameter from the dictionary "parameters" by using `parameters[".."]`. 2. Implement Forward Propagation. Compute `Z` multiplying arrays `W`, `X` and adding vector `b`. Set the prediction array A equal to Z.

[7]:
```
# GRADED FUNCTION: forward_propagation

def forward_propagation(X, parameters):
    """
    Argument:
    X -- input data of size (n_x, m), where n_x is the dimension input (in our␣
 ↪example is 2) and m is the number of training samples
    parameters -- python dictionary containing your parameters (output of␣
 ↪initialization function)

    Returns:
    Y_hat -- The output of size (1, m)
    """
    # Retrieve each parameter from the dictionary "parameters".
    W = parameters["W"]
    b = parameters["b"]

    # Implement Forward Propagation to calculate Z.
    ### START CODE HERE ### (~ 2 lines of code)
    Z = W @ X
    Y_hat = Z
    ### END CODE HERE ###


    return Y_hat
```

[6]:
```
w3_unittest.test_forward_propagation(forward_propagation)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-6432889118c3> in <module>
----> 1 w3_unittest.test_forward_propagation(forward_propagation)

NameError: name 'w3_unittest' is not defined
```

### 3.5 Defining the cost function

The cost function used to traing this model is

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

The next implementation is not graded.

[ ]:
```
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.shape[1]

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

    return cost
```

### Exercise 6

Now you're ready to implement your neural network. The next function will implement the training process and it will return the updated parameters dictionary where you will be able to make predictions.

[ ]:
```
# GRADED FUNCTION: nn_model

def nn_model(X, Y, num_iterations=1000, print_cost=False):
    """
    Arguments:
    X -- dataset of shape (n_x, number of examples)
    Y -- labels of shape (1, number of examples)
    num_iterations -- number of iterations in the loop
    print_cost -- if True, print the cost every iteration

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

    n_x = X.shape[0]

    # Initialize parameters
    parameters = utils.initialize_parameters(n_x)

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

        ### START CODE HERE ### (~ 2 lines of code)
        # 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)
        ### END CODE HERE ###


        # Parameters update.
        parameters = utils.train_nn(parameters, Y_hat, X, Y, learning_rate = 0.
↪001)

        # Print the cost every iteration.
        if print_cost:
            if i%100 == 0:
                print ("Cost after iteration %i: %f" %(i, cost))

    return parameters
```

[ ]:
```
w3_unittest.test_nn_model(nn_model)
```

### 3.6 - Training the neural network

Now let's load a dataset to train the neural network.

[ ]:
```
df = pd.read_csv("data/toy_dataset.csv")
```

[ ]:
```
df.head()
```

Let's first turn the data into a numpy array that we can pass to our function.

[ ]:
```
X = np.array(df[['x1','x2']]).T
Y = np.array(df['y']).reshape(1,-1)
```

Run the next block to update the parameters dictionary with the fitted weights.

[ ]:
```
parameters = nn_model(X,Y, num_iterations = 5000, print_cost= True)
```

## 4 - Make your predictions!

Now that you have the fitted parameters, you are able to predict any value with your neural network! You just need to perform the following computation:

$$Z = WX + b$$

Where W and b are in the parameters dictionary.

### Exercise 7

Now you will make the predictor function. It will input a parameters dictionary, a set of points X and output the set of predicted values.

```
[ ]: # GRADED FUNCTION: predict

     def predict(X, parameters):

         W = parameters['W']
         b = parameters['b']

         Z = np.dot(W, X) + b

         return Z
```

```
[ ]: y_hat = predict(X,parameters)
```

```
[ ]: df['y_hat'] = y_hat[0]
```

Now let's check some predicted values versus the original ones:

```
[ ]: for i in range(10):
         print(f"(x1,x2) = ({df.loc[i,'x1']:.2f}, {df.loc[i,'x2']:.2f}): Actual␣
      ↪value: {df.loc[i,'y']:.2f}. Predicted value: {df.loc[i,'y_hat']:.2f}")
```

Pretty good, right? Congratulations! You have finished the W3 assignment!