Deep Learning Specialization by deeplearning.ai

Course 1- Neural Networks and Deep Learning

Week 1- Introduction to Deep Learning

- AI is the new Electricity
- Electricity had once transformed countless industries: transportation, manufacturing, healthcare, communications and more.
- AI will now bring about an equally big transformation

1.1. What is a neural network ?

Neural networks are a model inspired by how the brain works. Similar to neurons in the brain, our "mathematical neurons" are also, intuitively, connected to each other; they take inputs (dendrites), do some simple computation on them and produce outputs (axons).

Let us start with a simple neural network:

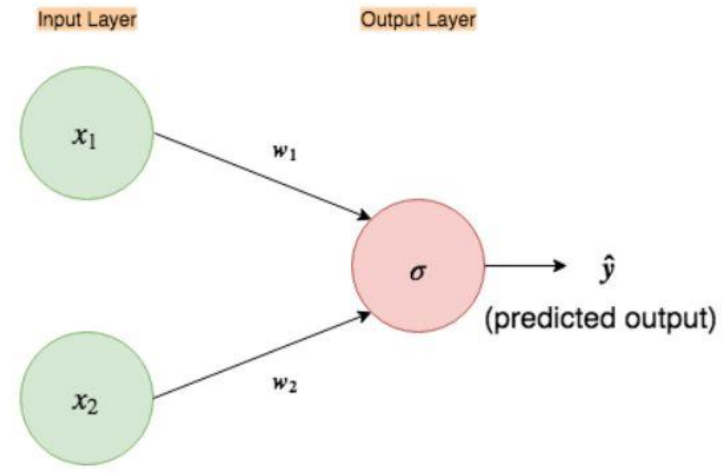


Fig 1. Simple input - output only neural network

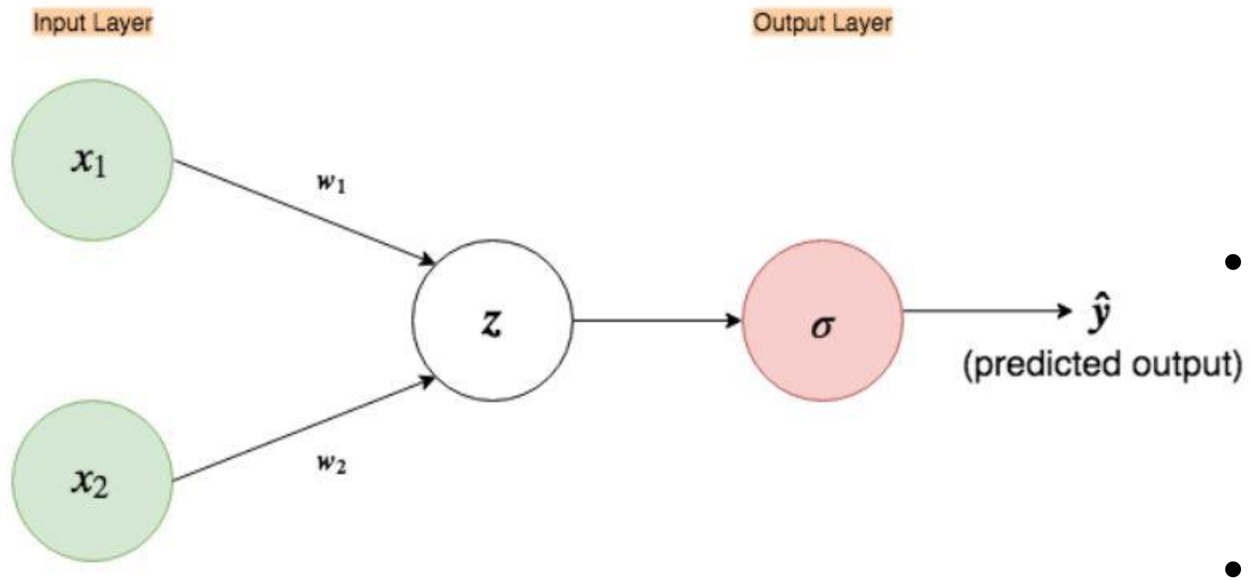
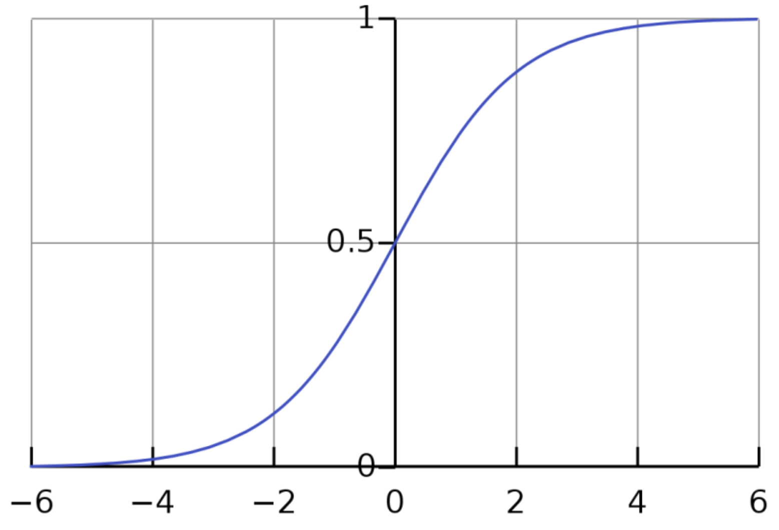


Fig 2. Expanded neural network

- x_1 x_2 are input nodes
- w_1 w_2 represent our weight vectors. Intuitively, these dictate how much influence each of the input features should have in computing the next node. You can think of them as the slope or gradient constant in a linear equation. Weights are the main values our neural network has to "learn". So initially, we will set them to random values and let the "learning algorithm" of our neural network decide the best weights that result in the correct output.
- $z = w_1x_1 + w_2x_2$, this node represents a linear function. Simply, it takes all the inputs coming to it and creates a linear equation out of them.
- The σ node takes the input and passes it through the sigmoid function.



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

Fig 3. Sigmoid activation function

Sigmoid is one of the many "activations functions" used in neural network. The job of an activation function is to change the input to a different range. For example, if $z > 2$ then, $\sigma(z) \approx 1$ and similarly, if $z < -2$ then, $\sigma(z) \approx 0$.

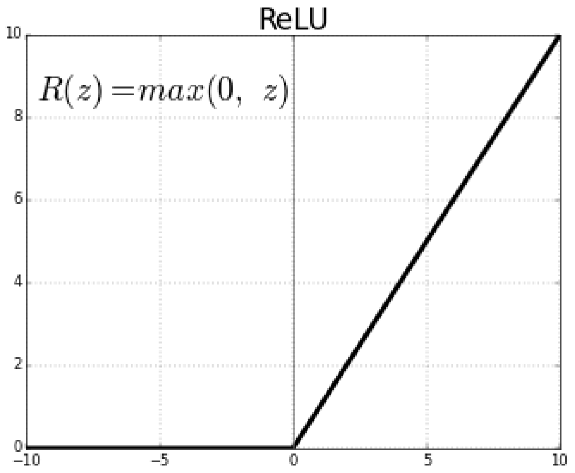


Fig4. ReLU activation function

ReLU stands for Rectified Linear Unit is another type of activation function. It is the most commonly activation function in neural networks, especially in CNNS.

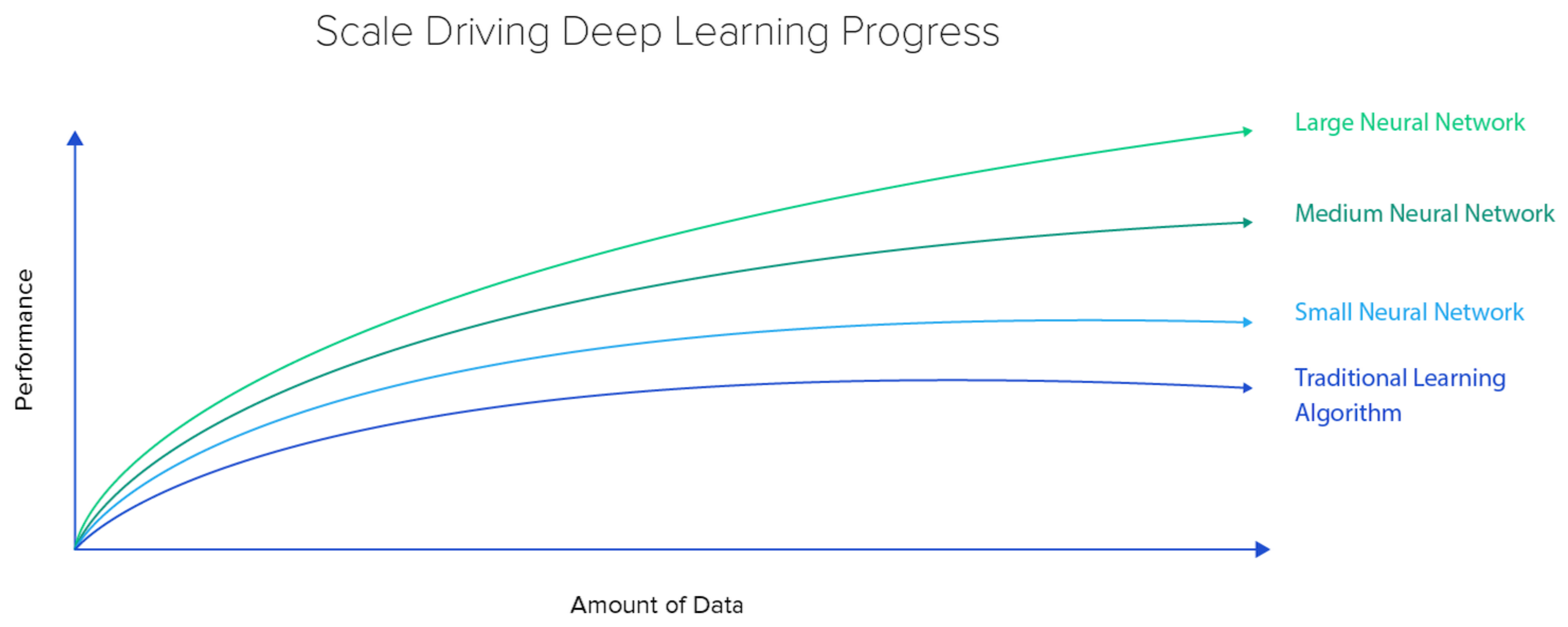
By switching from the sigmoid function to the ReLU function, we help the gradient descent run much faster.

1.2. Supervised Learning with Neural Networks

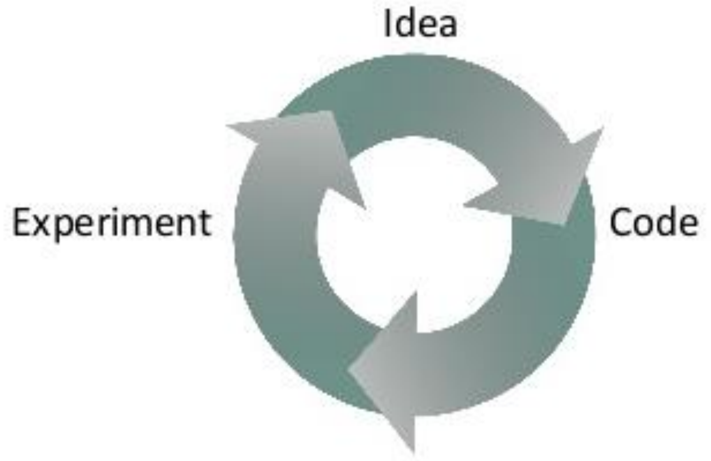
Input (x)	Output (y)	Application	DL Algorithm
Home features	Price	Real estate	Standard Neural Network
Ad, user info	Click on ad? (0/1)	Online advertising	Standard Neural Network
Image	Object (1,……., 1000)	Photo tagging	Convolutional Neural Network
Audio	Text transcript	Speech recognition	Recurrent Neural Network
English	Chinese	Machine translation	Recurrent Neural Network
Image, Radar Info	Position of other cars	Autonomous driving	Cluster/ Hybrid

Structured data: organized data in a table (excel sheet)
Unstructured data: audio, images, text, here the features might be the pixel values in an image or the individual words in a piece of text.

1.3. Why is Deep Learning taking off?



Deep Learning is taking off due to a large amount of data available through the digitization of the society, faster computation and innovation in the development of neural network algorithm.
The sources of data can be digital devices, inexpensive cameras, sensors in IoT
High Performance = Large NN (large amount of units, parameters, lot of connections) + Large amount of data



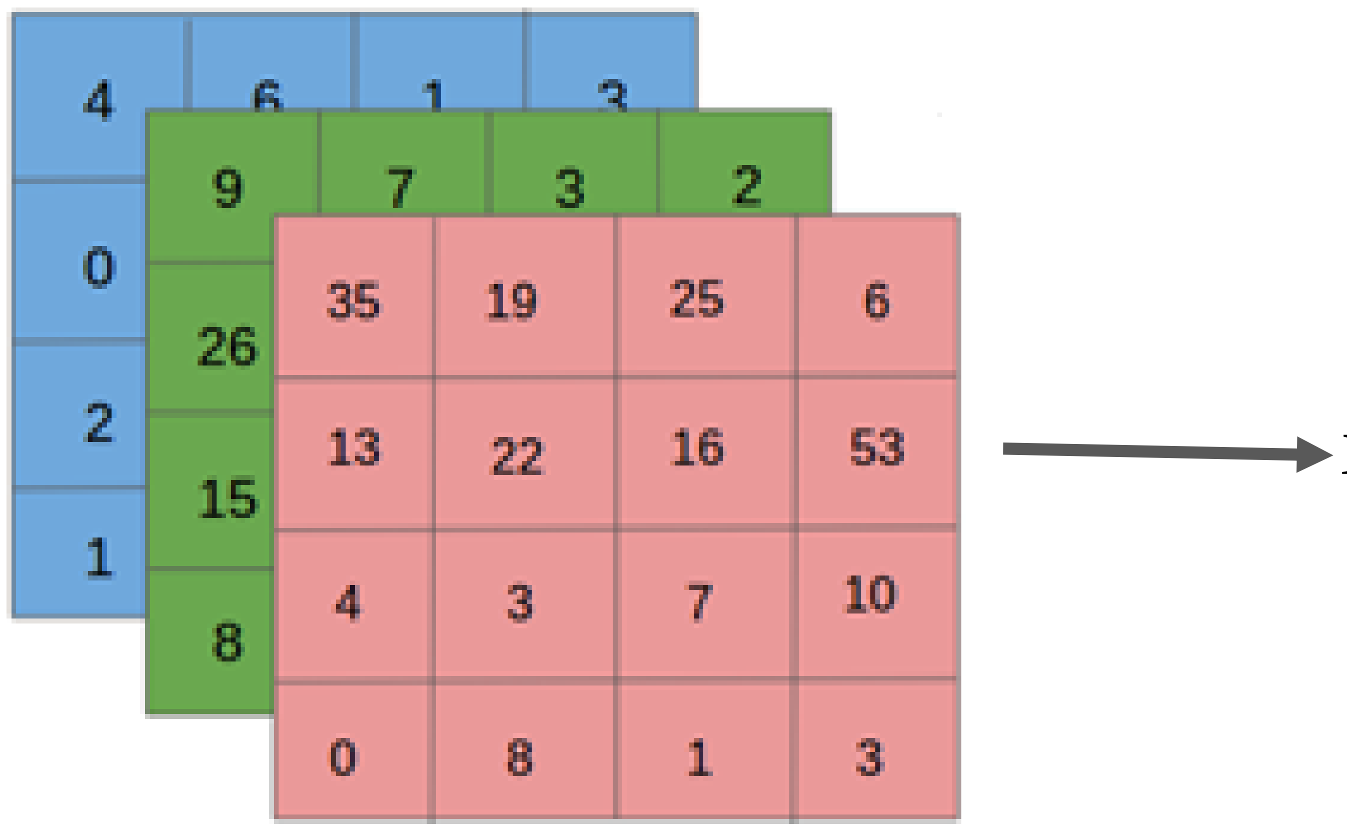
This loop explains the second reason why we need fact computation.

Week 2- Neural Networks Basics

2.1. Binary Classification

Logistic regression is an algorithm for binary classification.
If we input an image, it outputs 1 (cat) vs 0 (non cat).
An image stored in a computer = 3 layers of pixels corresponding to the RGB color channels of this image.
If we have 64x64 image then we have 3 64x64 matrices corresponding to the red, green, blue pixel intensity values for you image.
To unroll all these pixels intensity values into Feature vector:

$$(x, y): x \epsilon R^{n_x} \;\&\; y \epsilon \{0,1\}$$



$$x = \begin{bmatrix} 35 \\ 19 \\ \vdots \\ \vdots \\ 9 \\ 7 \\ \vdots \\ \vdots \\ 4 \\ 6 \\ \vdots \\ \vdots \end{bmatrix}$$

The dimension of the input feature x is :

$$n_x = 64 \; x \; 64 \; x \; 3 = 12288$$

Notation

$$\begin{array}{cc} X = \begin{bmatrix} | & | & & | \\ X^{(1)} & X^{(2)} & \cdots\cdots\cdots & X^{(m)} \\ | & | & & | \end{bmatrix} & \updownarrow n_x \\ \xleftrightarrow{\quad\quad m \quad\quad} & \end{array}$$

$X \in R^{n_x \; x \; m}$

X.shape $= (n_x$, m)

$$Y = [y^{(1)} \quad \ldots \quad \ldots \quad \ldots \quad y^{(m)}]$$

$Y \in R^{1 \; x \; m}$

Y.shape $=$ (1, m)

$(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)})$: first training example

m: number of training examples

2.2. Logistic Regression

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

x ϵ R^{n_x}

Parameters: w ϵ R^{n_x} & b ϵ R

Output: $\hat{y} = \sigma(w^T x + b),$ $(z = w^T x + b)$

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

- ➢ If z is large $\sigma(z) = \frac{1}{1+0} = 1$
- ➢ If z is small $\sigma(z) = \frac{1}{1+\infty} \approx 0$

2.3. Logistic Regression Cost Function

To train the parameters w and b of the logistic regression model you need to define a cost function.

$$\hat{y}^{(i)} = \sigma\left(w^T x^{(i)} + b\right) \; for\ the\ i^{th} training\ example$$

Training set

$$Given \left\{\left(x^{(1)}, y^{(1)}\right), \ldots \ldots, \left(x^{(m)}, y^{(m)}\right)\right\}, want\ \hat{y}^{(i)} \approx y^{(i)}$$

$$\boldsymbol{Loss\ (error)function}: L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

is used to measure how good is y. But we will use another loss function because this function (squared error) makes gradient descent not work well.

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

We need to be as small as possible

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

- $If\ y = 1\text{: } L(\hat{y}, y) = -\log(\hat{y})$
- $If\ y = 0\text{: } L(\hat{y}, y) = -\log(1 - \hat{y})$

$\boldsymbol{Cost\ Function}$: $J(w, b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$ = - $\frac{1}{m}\sum_{i=1}^{m}(y\log(\hat{y}) + (1-y)\log(1-\hat{y}))$

The loss function computes the error for a single training example, the cost function is the average of the loss function of the entire training set.

The loss function measures how well your algorithm outputs $\hat{y}^{(i)}$ on each of the training examples compared to the ground true label $y^{(i)}$ on each of the training examples.

The cost function measures how well your parameters w and b are doing on the training set.

2.4. Gradient Descent

The gradient descent algorithm is used to train or to learn, the parameters x and b on your training set.

Want to find w, b that minimize J(w, b)

- Initial point on the graph J(w, b).
- Gradient descent takes a step in the steepest downhill direction until finally converge to this global optimum.

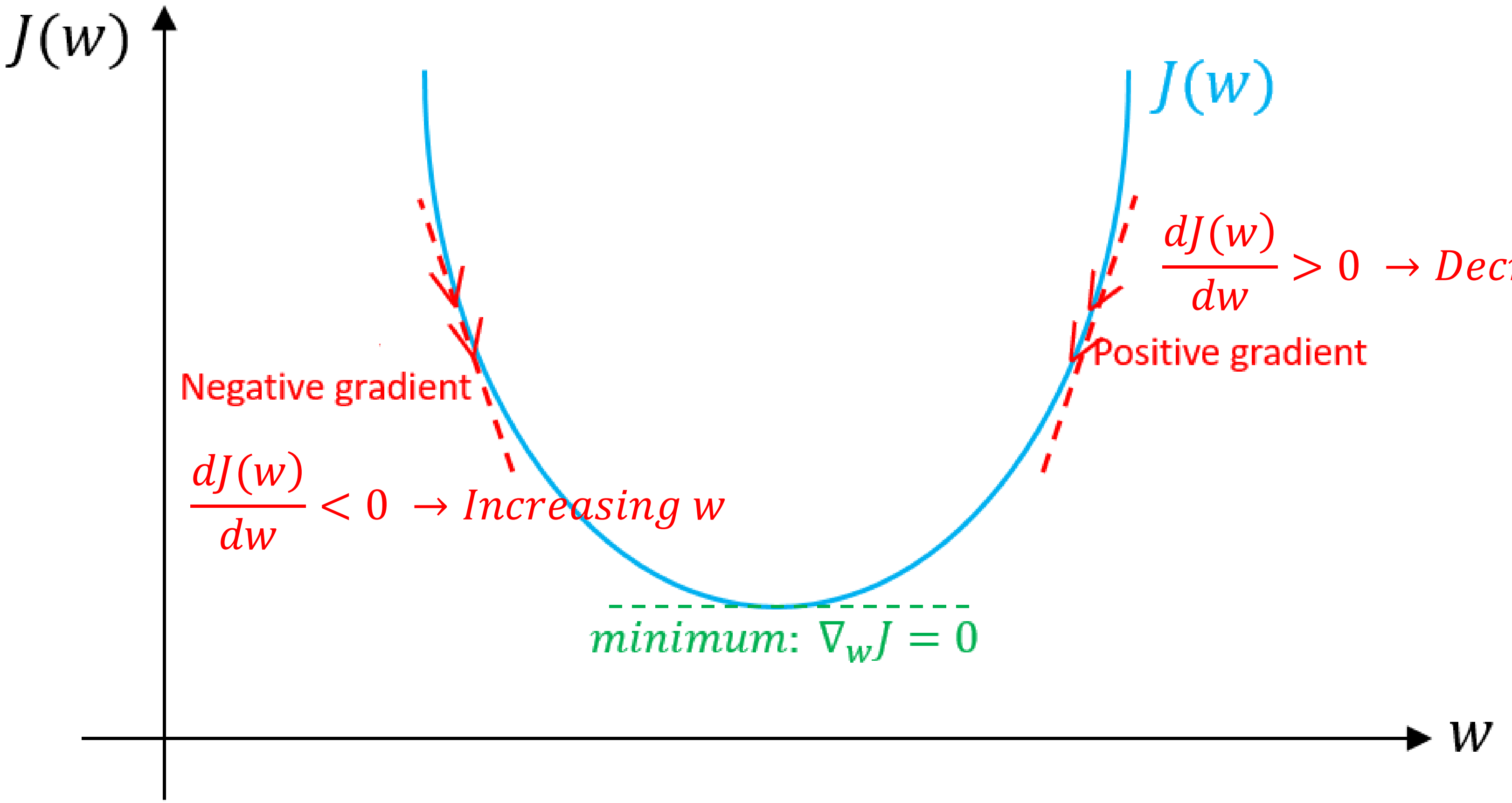
More formally, gradient descent is an optimization algorithm used to minimize some function by iteratively moving the direction of the steepest descent as defined by the negative of the gradient. In neural networks, we use gradient descent to update the weights of our model.

$$Repeat \{$$
$$w := w - \propto \frac{dJ(w)}{dw}$$
$$\}$$

The change you want to make to the parameter w

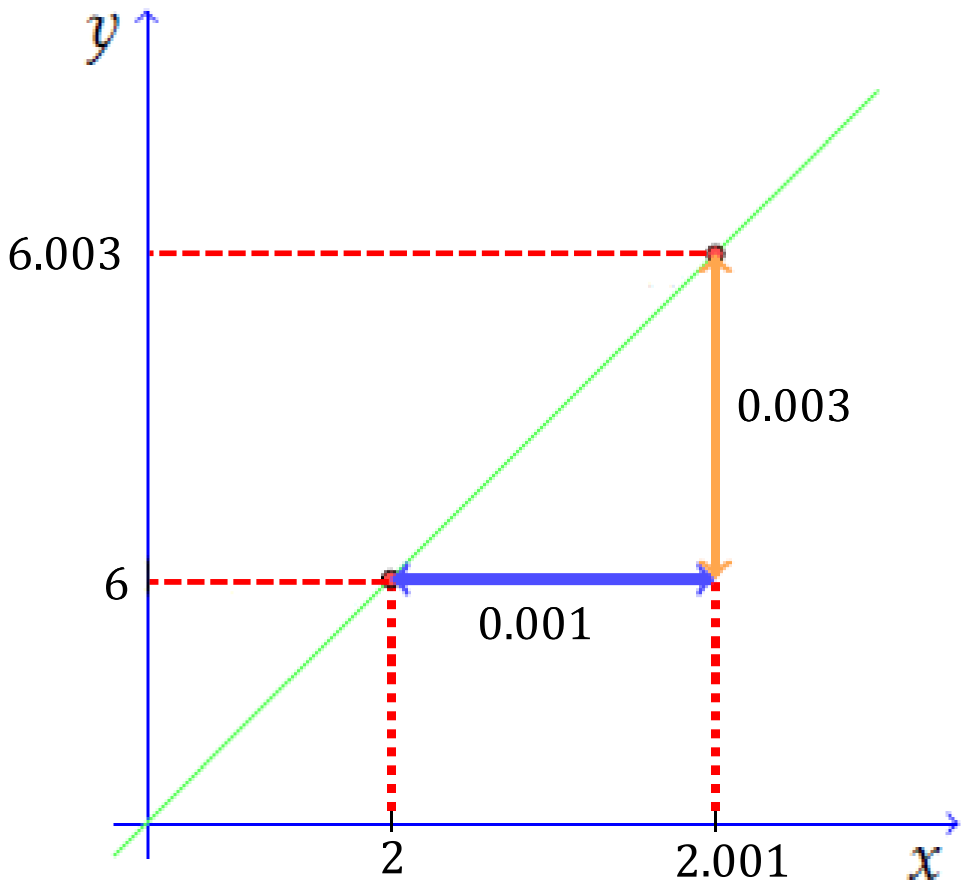
Slope of the tangent on J(w)

Learning rate: it controls how big a step we take on each iteration or gradient descent



$J(w)$
$J(w)$
$\frac{dJ(w)}{dw} > 0 \rightarrow Decreasing\ w$
Positive gradient
Negative gradient
$\frac{dJ(w)}{dw} < 0 \rightarrow Increasing\ w$
$minimum: \nabla_w J = 0$
w

2.5. Intuition about derivatives



$$a = 2 \rightarrow f(a) = 6$$

$$a = 2.001 \rightarrow f(a) = 6.003$$

Slope (derivative) of f(a) at a = 2 is 3

$$\frac{height}{width} = \frac{0.003}{0.001} = 3 = \frac{df(a)}{da} = \frac{d}{da}f(a)$$

2.6. Logistic Regression Gradient Descent

$$\begin{tikzcd}
x_1 \arrow[ddr] & & & \\
w_1 \arrow[dr] & & & \\
x_2 \arrow[r] & \boxed{z = w_1x_1 + w_2x_2 + \mathrm{b}} \arrow[r] & \boxed{\hat{y} = a = \sigma(z)} \arrow[r] & \boxed{L(a,y)} \\
w_1 \arrow[ur] & & & \\
b \arrow[uur] & & &
\end{tikzcd}$$

$$\frac{dL}{dz} = dz = \frac{dL}{da}\ x\ \frac{da}{dz} = a - y$$

$\boldsymbol{Step\ 1: \frac{dL}{da}}$

$$L = -(y\log(a) + (1-y)\log(1-\mathrm{a}))$$

$$\frac{dL}{da} = -y\ x\frac{1}{a} - (1-y)\ x\ \frac{1}{1-a}\ x - 1$$

Remember that there is an additional -1 in the last term when we take the derivative of (1 – a) with respect to a.

$$\frac{dL}{da} = \frac{-\mathrm{y}}{a} + \frac{1-y}{1-a} = \frac{-y+ay+a-ay}{\mathrm{a}(1-a)} = \frac{a-y}{\mathrm{a}(1-a)} = \mathrm{da}$$

$\boldsymbol{Step\ 2}$: $\boldsymbol{\frac{da}{dz}}$

$$\frac{da}{dz} = \frac{d}{dz}\sigma(z)$$

The derivative of a sigmoid has the form:

$$\frac{d}{dz}\sigma(z) = \sigma(z) \text{ x } (1 - \sigma(z))$$

Recall that $\sigma(z)$ = a, because we defined “a”, the activation, as the output of the sigmoid activation function.

$$\frac{da}{dz} = a(1 - a)$$

$\boldsymbol{Step\ 3}$: $\boldsymbol{\frac{dL}{dz}}$

$$\frac{dL}{dz} = dz = \frac{dL}{da} \; x \; \frac{da}{dz} = \frac{a - y}{\mathrm{a}(1-a)} \; x \; a\,(a-y) = a - y$$

$$\frac{dL}{dw_1} = dw_1 = x_1 dz \qquad w_1 \coloneqq w_1 - \propto dw_1$$

$$\frac{dL}{dw_2} = dw_2 = x_2 dz \qquad w_2 \coloneqq w_2 - \propto dw_2$$

$$db = dz \qquad b \coloneqq b - \propto db$$

2.7. Vectorization

```python
import numpy as np
a = np.array([1,2,3,4])
print(a)

import time
a = np.random.rand(1000000)
b = np.random.rand(1000000)
tic = time.time()
c = np.dot(a,b)
toc = time.time()
print("Vectorized version:" + str(1000*(toc -tic)) + "ms")
```

Vectorized version: 0.9982585906982422 ms

```
import numpy as np
a = np.array([1,2,3,4])
print(a)

import time
c = 0
a = np.random.rand(1000000)
b = np.random.rand(1000000)
tic = time.time()
for i in range(1000000):
    c += a[i]*b[i]
    toc = time.time()
print("For Loop" + str(1000*(toc -tic)) + "ms")
```

```
For Loop 622.3413944244385 ms
```

We can conclude that the vectorised implementation (for loop) made the computation much faster. So whenever possible, avoid explicit for loops.

2.7. Vectorising Logistic Regression

$z^{(1)} = w^T x^{(1)} + b$ $\qquad$ $z^{(2)} = w^T x^{(2)} + b$ $\qquad$ $z^{(3)} = w^T x^{(3)} + b$

$a^{(1)} = \sigma(z^{(1)})$ $\qquad$ $a^{(2)} = \sigma(z^{(2)})$ $\qquad$ $a^{(3)} = \sigma(z^{(3)})$

1 x m

$$\boldsymbol{Z} = [\boldsymbol{z}^{(\boldsymbol{1})} \quad \cdots \quad \cdots \quad \cdots \quad \boldsymbol{z}^{(\boldsymbol{m})}]$$
$$= w^T X + [b \quad \cdots \quad \cdots \quad \cdots \quad b]$$
$$= [\, w^T x^{(1)} + b \quad \cdots \quad \cdots \quad \cdots \quad w^T x^{(m)} + b]$$

In Python: Z = np.dot (W.T, X) + b

$$\boldsymbol{A} = [\boldsymbol{a}^{(\boldsymbol{1})} \quad \cdots \quad \cdots \quad \cdots \quad \boldsymbol{a}^{(\boldsymbol{m})}] = \boldsymbol{\sigma}(\mathbf{Z})$$

2.8. Vectorising Logistic Regression's Gradient Computation

$dz^{(1)} = a^{(1)} - y^{(1)}$ $\quad$ $dz^{(2)} = a^{(2)} - y^{(2)}$

$A = [a^{(1)} \quad \dots \quad \dots \quad \dots \quad a^{(m)}]$ $\quad$ $Y = [y^{(1)} \quad \dots \quad \dots \quad \dots \quad y^{(m)}]$

1 x m

$\boldsymbol{dZ} = [\boldsymbol{dz}^{(\mathbf{1})} \quad \dots \quad \dots \quad \dots \quad \boldsymbol{dz}^{(\boldsymbol{m})}]$

$= [a^{(1)} - y^{(1)} \quad \dots \quad \dots \quad \dots \quad a^{(m)} - y^{(m)}]$

<u>Vectorized Implementation</u>

$Z = w^T X + b$= np.dot(W.T, X) + b

$A = \sigma(Z)$

$dZ = A - Y$

$dW = \frac{1}{m}\, x\, dZ^{(T)}$

$db = np.sum(dZ)$

$w = w - \propto dw$

$b = b - \propto db$

2.9. Broadcasting in Python

The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorising array operations so that looping occurs in C instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations.

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In this simplest case, the two arrays must have exactly the same shape.

```
import numpy as np
a = np.array([1,2,3])
b = np.array([2,2,2])
print(a*b)
```

```
[2 4 6]
```

NumPy’s broadcasting rule relaxes this constraint when the array’s shapes meet certain constraints. The simplest broadcasting example occurs when an array and a scalar value are combined in an operation:

```python
import numpy as np
a = np.array([1,2,3])
b = 2
print(a*b)
```

```
[2 4 6]
```

The result is equivalent to the previous example where b was an array. We can think of the scalar b being stretched during the arithmetic operation into an array with the same shape as a. The new elements in b are simply copies of the original scalar. The stretching analogy is only conceptual. NumPy is smart enough to use the original scalar value without actually making copies, so that broadcasting operations are as memory and computationally efficient as possible.

Assignment 1: Logistic Regression with a Neural Network mindset

You will build a logistic regression classifier to recognize cats. This assignment will step you through how to do this with a Neural Network mindset, and so will also hone your intuitions about deep learning.

You will learn to:

- Build the general architecture of a learning algorithm, including:
- Initializing parameters
- Calculating the cost function and its gradient
- Using an optimization algorithm (gradient descent)
- Gather all three functions above into a main model function, in the right order.

1. Packages

The packages needed for this assignment are:

- numpy is the fundamental package for scientific computing with python
- h5py is a common package to interact with a dataset that is stored on an H5 file.
- matplotlib is a famous library to plot graphs in Pyhton
- PIL and scipy are used here to test your model with your own picture at the end

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
from lr_utils import load_dataset
```

2. Overview of the Problem set

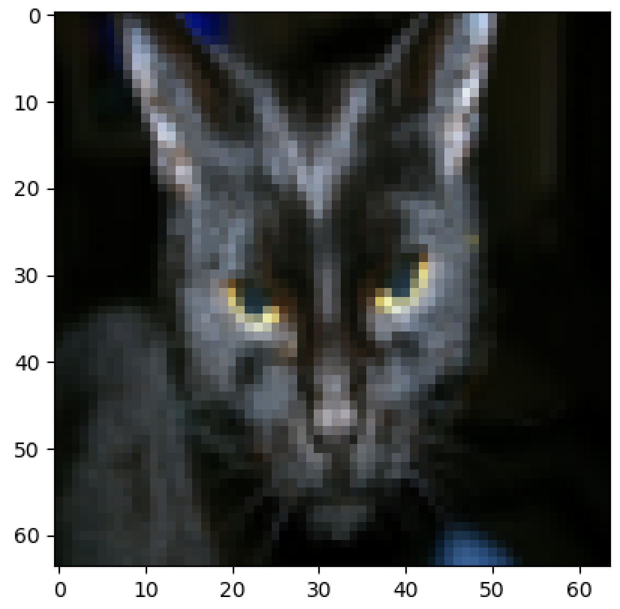
Problem Statement: You are given a dataset (“data.h5”) containing:

- A training set of m_train images labeled as cat (y = 1) or non-cat (y = 0)
- A test set of m_test images labeled as cat or non-cat
- Each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB). Thus, each image is square (height = num_px) and (width = num_px)

```
# Loading the data (cat/non-cat)
train_dataset = h5py.File('C:/Users/HP/Downloads/train_catvnoncat.h5', 'r')
train_set_x_orig = np.array(train_dataset["train_set_x"][:])
train_set_y_orig = np.array(train_dataset["train_set_y"][:])

test_dataset = h5py.File('C:/Users/HP/Downloads/test_catvnoncat.h5', 'r')
test_set_x_orig = np.array(test_dataset["test_set_x"][:])
test_set_y_orig = np.array(test_dataset["test_set_y"][:])

classes = np.array(test_dataset["list_classes"][:])

train_set_y = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
test_set_y = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))
```

We added "_orig" at the end of image datasets (train and test) because we are going to preprocess them. After preprocessing, we will end up with train_set_x and test_set_x (the labels of train_set_y, and test_set_y do not need preprocessing)

```
# Example of a picture
index = 25
plt.imshow(train_set_x_orig[index])
print("y = " + str(train_set_y[:, index]) + " , it is a '"+
classes[np.squeeze(train_set_y[:,index])].decode("utf-8")+"'picture.")
```



y = [1] , it is a 'cat' picture.

Many software bugs in deep learning come from having matrix/vector dimensions that does not fit. If you can keep your matrix/vector dimensions straight you will go a long way toward eliminating many bugs.

```python
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]
print("Number of training examples: m_train = " + str(m_train))
print("Number of test examples: m_test = " + str(m_test))
print("Height/Width of each image: num_px = " + str(num_px))
```

```
Number of training examples: m_train = 209
Number of test examples: m_test = 50
Height/Width of each image: num_px = 64
```

For convenience, you should now reshape images of shape (num_px, num_px, 3) in a numpy array of shape (num_px*num_px*3, 1). After this, our training (and test) dataset is a numpy array where each column represents a flattened image. There should be m_train (respectively m_test) columns.

```
# Reshape the training and test examples
train_set_x_flatten =
train_set_x_orig.reshape(train_set_x_orig.shape[1]*train_set_x_orig.shape[1]*3 , -1)
test_set_x_flatten =
test_set_x_orig.reshape(test_set_x_orig.shape[1]*test_set_x_orig.shape[1]*3 , -1)
print ("train_set_x_flatten shape: " + str(train_set_x_flatten.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x_flatten shape: " + str(test_set_x_flatten.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
print ("sanity check after reshaping: " + str(train_set_x_flatten[0:5,0]))
```

```
train_set_x_flatten shape: (12288, 209)
train_set_y shape: (1, 209)
test_set_x_flatten shape: (12288, 50)
test_set_y shape: (1, 50)
sanity check after reshaping: [17 71 49 38 70]
```

To represent color images, the red, green, and blue channels (RGB) must be specified for each pixel, and so the pixel value is actually a vector of three numbers ranging from 0 to 255.

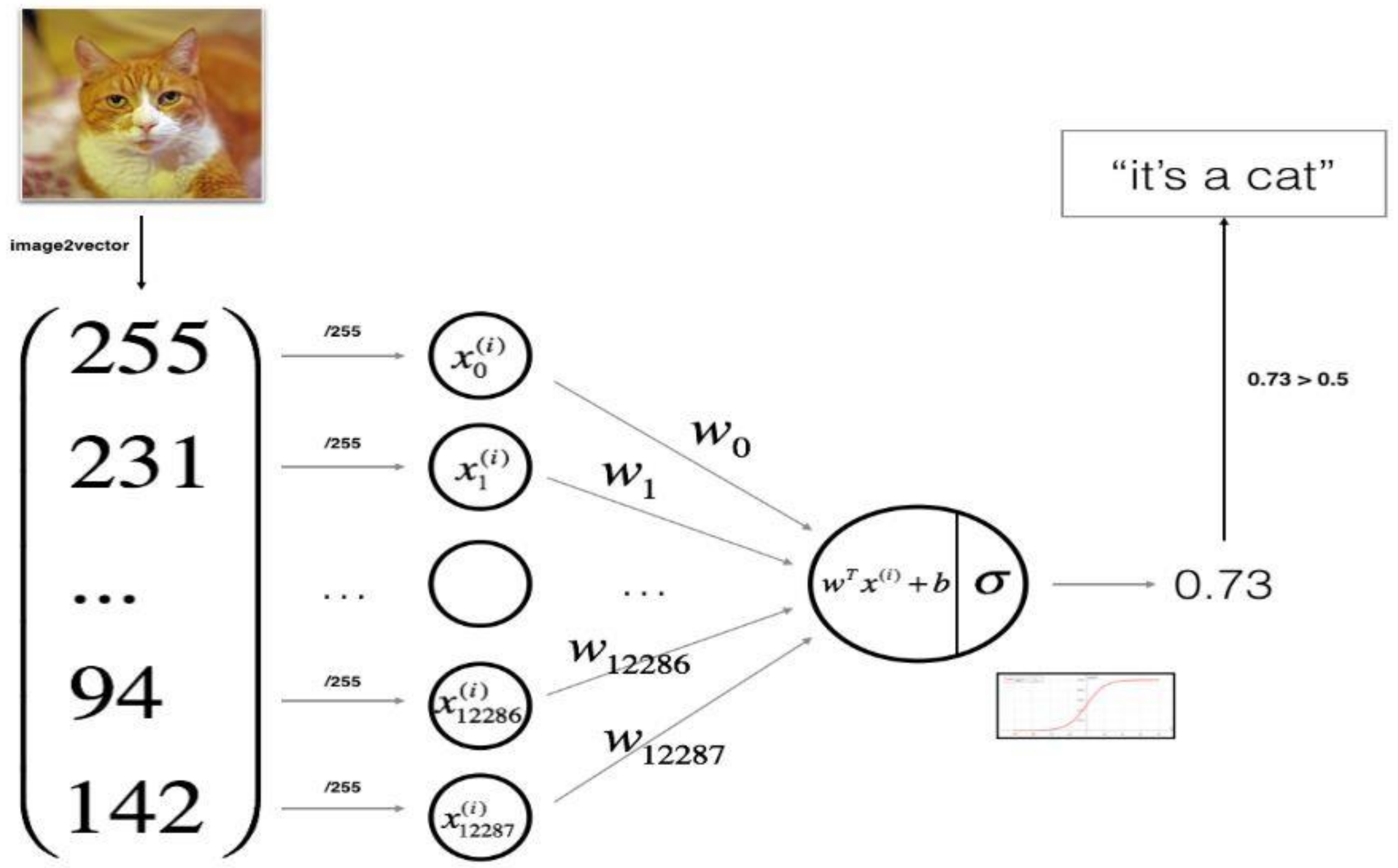
One common preprocessing step in machine learning is to center and standardize your dataset, meaning that you subtract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array. But for pictures datasets, it is simpler and more convenient and works almost as well as to divide every row of the dataset by 255 (the maximum value of a pixel channel).

```
# Standardize our dataset
train_set_x = train_set_x_flatten/255
test_set_x = test_set_x_flatten/255
```

You need to remember: common steps for pre-processing a new dataset are:

- Figure out the dimensions and shapes of the problem (m_train, m_test, num_px,….)
- Reshape the datasets such that each example is now a vector of size (num_px*num_px*3, 1).
- Standardize the data

3. General Architecture of the learning algorithm



Mathematical expression of the algorithm:

For one example $x^{(i)}$:

$$z^{(i)} = w^T x^{(i)} + b \tag{1}$$
$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)}) \tag{2}$$
$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1-y^{(i)}) \log(1-a^{(i)}) \tag{3}$$

The cost is then computed by summing over all training examples:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}) \tag{6}$$

Key steps: In this exercise, you will carry out the following steps:

```
    - Initialize the parameters of the model
    - Learn the parameters for the model by minimizing the cost
    - Use the learned parameters to make predictions (on the tes
t set)
    - Analyse the results and conclude
```

4. Building the parts

The main steps for building a Neural Network are:

1. Define the model structure (such as number of input features)
2. Initialize the model’s parameters
3. Loop:

- Calculate current loss (forward propagation)
- Calculate current gradient descent (backward propagation)
- Update parameters (gradient descent)

You often build 1-3 separately and integrate them into one function we call model().

4.1. Helper functions

```
# sigmoid
def sigmoid(z):
    s = 1. / (1. + np.exp(-z))
    return s
print("sigmoid([0,2]) = " + str(sigmoid(np.array([0,2]))))
```

```
sigmoid([0,2]) = [0.5        0.88079708]
```

4.2. Initializing parameters

This function creates a vector of zeros of shape (dim, 1) for w and initializes b to 0.

Argument:
dim -- size of the w vector we want (or number of parameters in this case)

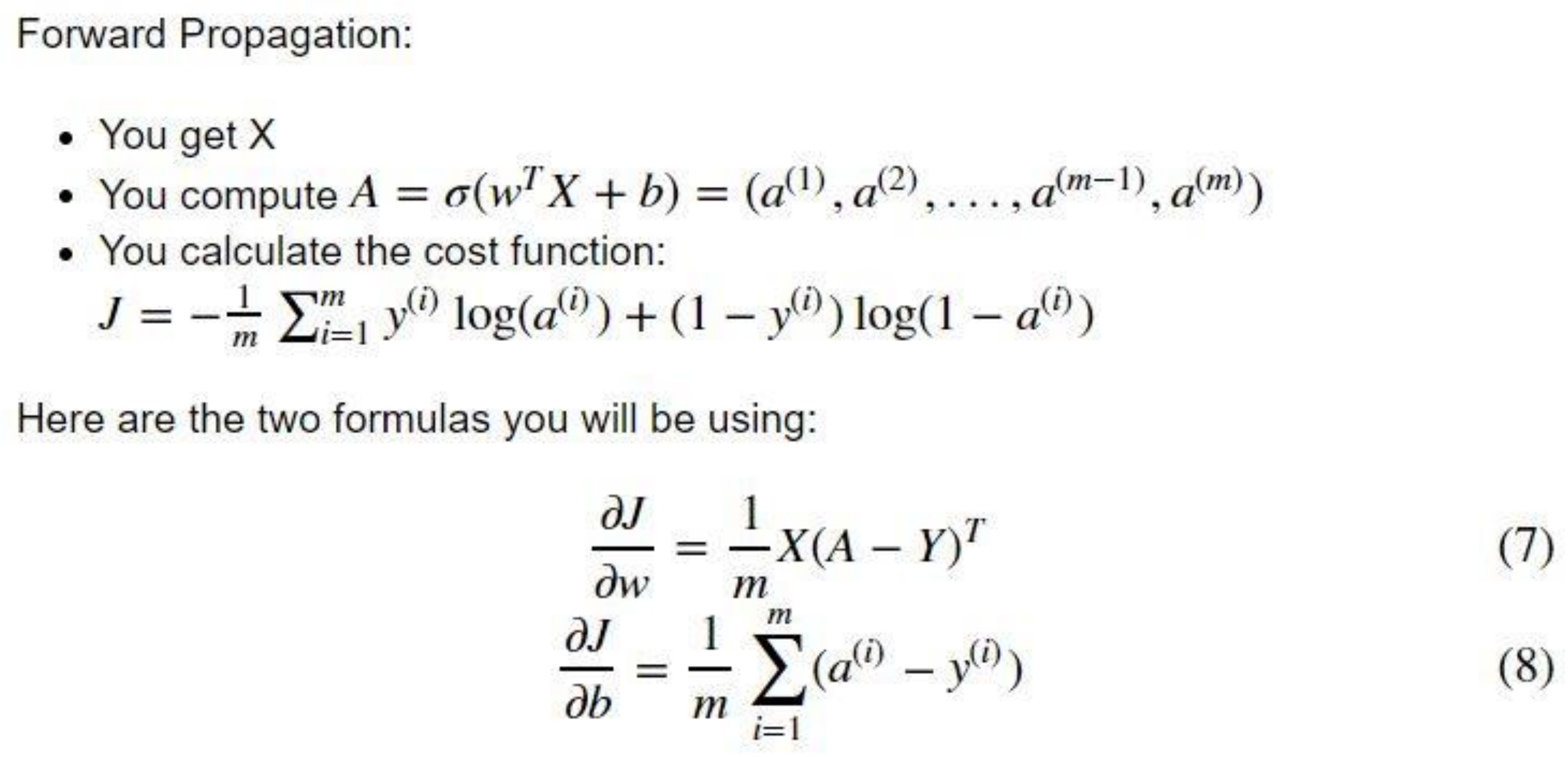
Returns:
w -- initialized vector of shape (dim, 1)
b -- initialized scalar (corresponds to the bias)

```python
# initialize_with_zeros
def initialize_with_zeros(dim):
    w = np.zeros((dim, 1))
    b = 0
    assert (w.shape == (dim, 1))
    assert (isinstance(b, float) or isinstance(b, int))
    return w, b
dim = 2
w, b = initialize_with_zeros(dim)
print("w = " + str(w))
print("b = " + str(b))
```

```
w = [[0.]
 [0.]]
b = 0
```

4.3. Forward and Backward propagation

Now that your parameters are initialized, you can do the “forward” and “backward” propagation steps for learning the parameters.



Forward Propagation:

- You get X
- You compute $A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, \ldots, a^{(m-1)}, a^{(m)})$
- You calculate the cost function: $J = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log(a^{(i)}) + (1-y^{(i)})\log(1-a^{(i)})$

Here are the two formulas you will be using:

$$\frac{\partial J}{\partial w} = \frac{1}{m}X(A-Y)^T \tag{7}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)}) \tag{8}$$

Implement the cost function and its gradient for the propagation explained above

Arguments:
w -- weights, a numpy array of size (num_px * num_px * 3, 1)
b -- bias, a scalar
X -- data of size (num_px * num_px * 3, number of examples)
Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1, number of examples)

Return:
cost -- negative log-likelihood cost for logistic regression
dw -- gradient of the loss with respect to w, thus same shape as w
db -- gradient of the loss with respect to b, thus same shape as b

Tips:
- Write your code step by step for the propagation. np.log(), np.dot()

```python
# propagate
def propagate(w,b, X, Y):
    m = X.shape[1]
    # Forward propagation (From X to COST)
    A = sigmoid(np.dot(w.T, X) + b)
    cost = -1/m * np.sum(np.dot(np.log(A), Y.T) + np.dot(np.log(1-A), (1-Y.T)))
    # Backward Propagation
    dw = 1/m*np.dot(X, (A-Y).T)
    db = 1/m*np.sum(A-Y)
    assert (dw.shape == w.shape)
    assert (db.dtype == float)
    cost = np.squeeze(cost)
    assert (cost.shape == ())
    grads = {"dw" : dw,
             "db" : db}
    return grads, cost
w, b, X, Y = np.array([[1.],[2.]]), 2., np.array([[1.,2.,-1.],[3.,4.,-3.2]]),
np.array([[1,0,1]])
grads, cost = propagate(w, b, X, Y)
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print ("cost = " + str(cost))
```

```
dw = [[0.99845601]
 [2.39507239]]
db = 0.00145557813678
cost = 5.801545319394553
```

4.4. Optimization

- You have initialized your parameters
- You are also able to compute a cost function and its gradient
- Now, you want to update the parameters using gradient descent

This function optimizes w and b by running a gradient descent algorithm

Arguments:
w -- weights, a numpy array of size (num_px * num_px * 3, 1)
b -- bias, a scalar
X -- data of shape (num_px * num_px * 3, number of examples)
Y -- true "label" vector (containing 0 if non-cat, 1 if cat), of shape (1, number of examples)
num_iterations -- number of iterations of the optimization loop
learning_rate -- learning rate of the gradient descent update rule
print_cost -- True to print the loss every 100 steps

Returns:
params -- dictionary containing the weights w and bias b
grads -- dictionary containing the gradients of the weights and bias with respect to the cost function
costs -- list of all the costs computed during the optimization, this will be used to plot the learning curve.

Tips:
You basically need to write down two steps and iterate through them:
1) Calculate the cost and the gradient for the current parameters. Use propagate().
2) Update the parameters using gradient descent rule for w and b.

```
def optimize (w, b, X, Y, num_iterations, learning_rate, print_cost =
False):
    costs = []
    for i in range(num_iterations):
        # cost and gradient calculation
        grads, cost = propagate(w, b, X, Y)
        # Retrieve the derivatives from grads
        dw = grads["dw"]
        db = grads["db"]
        # update rule
        w = w - learning_rate*dw
        b = b - learning_rate*db
        # Record the costs
        if i%100 == 0:
            costs.append(cost)
        # print the cost every 100 training iterations
        if print_cost and i%100 == 0:
            print("Cost after iteration %i: %f" %(i, cost))
```

```python
    params = {"w": w,
              "b": b}
    grads = {"dw": dw,
             "db": db}
    return params, grads, costs
params, grads, costs = optimize(w, b, X, Y, num_iterations= 100,
learning_rate = 0.009, print_cost = False)
print ("w = " + str(params["w"]))
print ("b = " + str(params["b"]))
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
```

```
w = [[0.19033591]
 [0.12259159]]
b = 1.9253598300845747
dw = [[0.67752042]
 [1.41625495]]
db = 0.21919450454067657
```

```python
# predict
def predict(w, b, X):
    m = X.shape[1]
    Y_prediction = np.zeros((1,m))
    w = w.reshape(X.shape[0], 1)
    # compute vector A predicting the probabilities of a cat being present
    A = sigmoid(np.dot(w.T, X)+ b)
    for i in range(A.shape[1]):
        # convert probabilities A[0,i] to actual predictions p[0,i]
        Y_prediction = np.where(A > 0.5, 1, 0)
    assert(Y_prediction.shape == (1,m))
    return Y_prediction
w = np.array([[0.1124579], [0.23106775]])
b = -0.3
X = np.array([[1.,-1.1,-3.2],[1.2,2.,0.1]])
print("predictions = " + str(predict(w,b,X)))
```

predictions = [[1 1 0]]

5. Merge all into a model

You need to remember: you have implemented several functions that:

- Initialize (w,b)
- Optimize the loss iteratively to learn parameters (w,b):
- Computing the cost and its gradient
- Updating the parameters using gradient descent
- Use the learned (w,b) to predict the labels for a given set of examples.

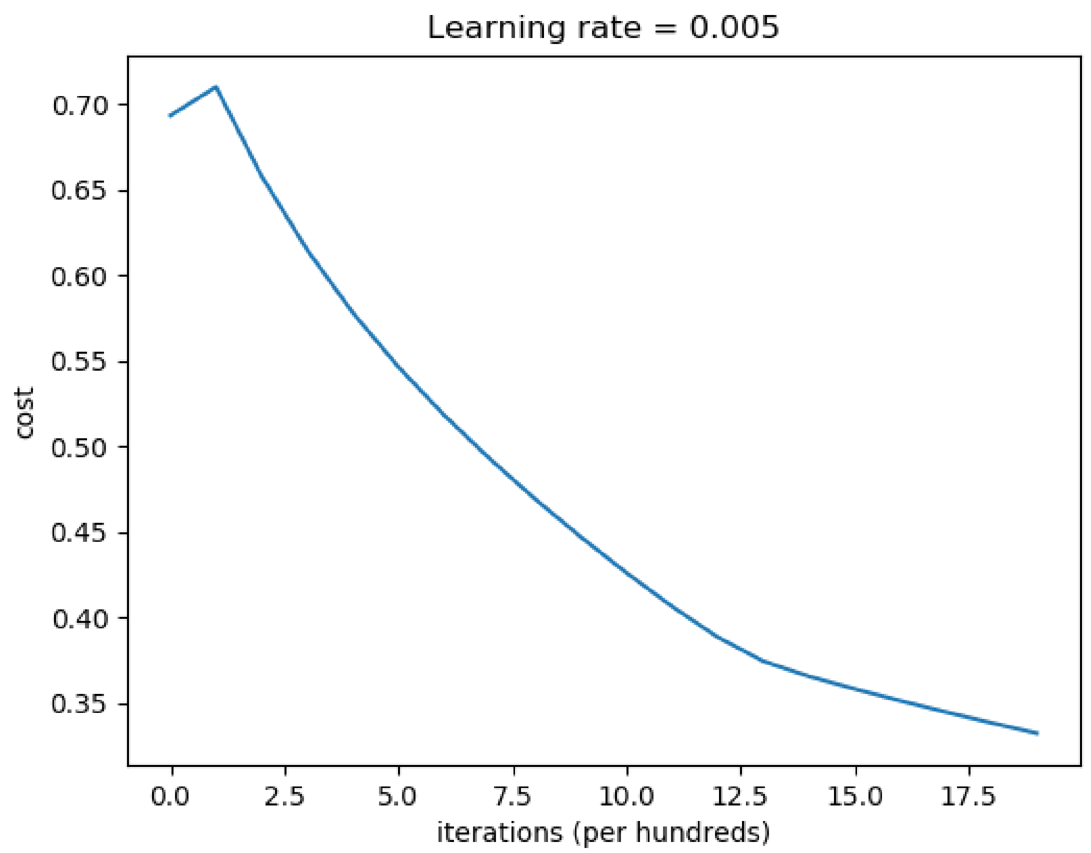
You will now see how the overall model is structured by outing together all the building blocks (functions implemented in the previous parts) together in the right order.

```
def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000,
learning_rate = 0.5, print_cost = False):
    # initialize parameters with zeros
    w, b = initialize_with_zeros(X_train.shape[0])
    # Gradient descent
    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, learning_rate,
print_cost = True)
    # Retrieve parameters w and b from dictionary parameters
    w = parameters["w"]
    b = parameters["b"]
    # Predict test/train set examples
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)
    # Print train/test Errors
    print("train accuracy:{} %".format(100 - np.mean(np.abs(Y_prediction_train - Y_train)) *100))
    print("test accuracy:{} %".format(100 - np.mean(np.abs(Y_prediction_test - Y_test)) * 100))
```

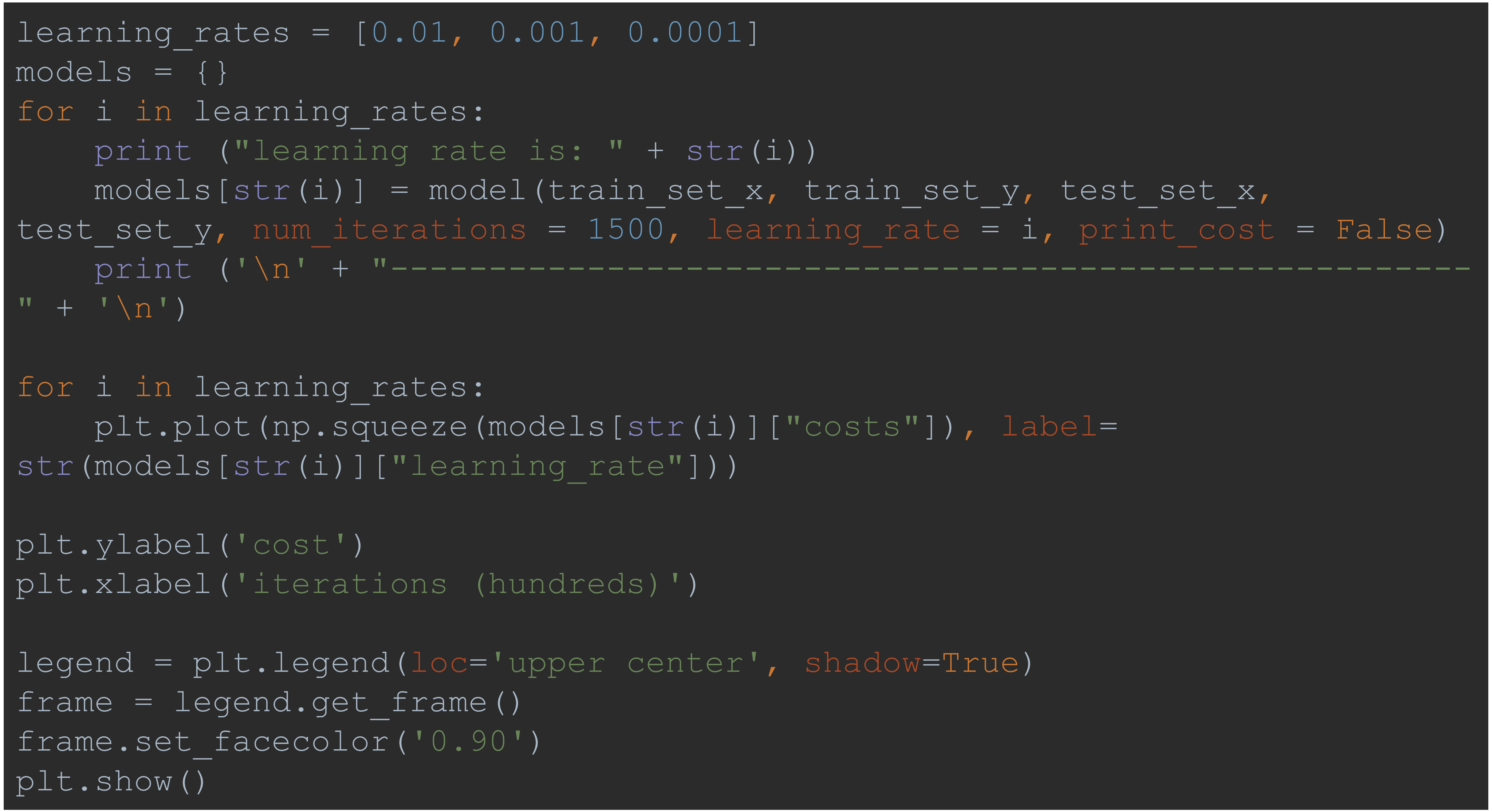
```python
    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train": Y_prediction_train,
         "w": w,
         "b": b,
         "learning_rate": learning_rate,
         "num_iterations": num_iterations}
    return d
d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000, learning_rate =
0.005, print_cost = True)
```

```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.709726
Cost after iteration 200: 0.657712
Cost after iteration 1800: 0.338704
Cost after iteration 1900: 0.332664
train accuracy: 91.38755980861244 %
test accuracy: 34.0 %
```

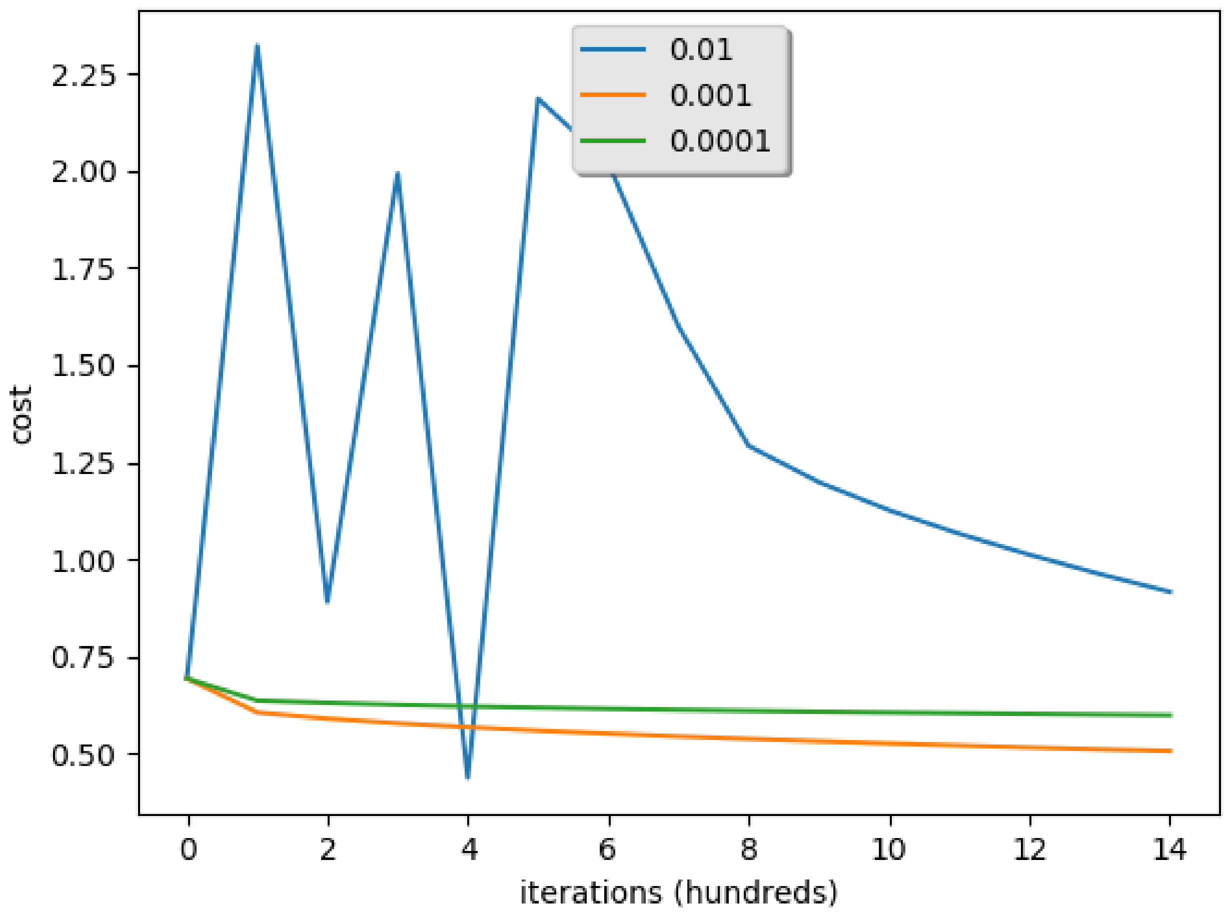
```
# Plot learning curve (with costs)
costs = np.squeeze(d['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate = " + str(d["learning_rate"]))
plt.show()
```



You can see the cost decreasing. It shows that the parameters are being learned. However, you see that you could train the model even more on the training set.



```
learning_rates = [0.01, 0.001, 0.0001]
models = {}
for i in learning_rates:
    print ("learning rate is: " + str(i))
    models[str(i)] = model(train_set_x, train_set_y, test_set_x,
test_set_y, num_iterations = 1500, learning_rate = i, print_cost = False)
    print ('\n' + "-------------------------------------------------------
" + '\n')

for i in learning_rates:
    plt.plot(np.squeeze(models[str(i)]["costs"]), label=
str(models[str(i)]["learning_rate"]))

plt.ylabel('cost')
plt.xlabel('iterations (hundreds)')

legend = plt.legend(loc='upper center', shadow=True)
frame = legend.get_frame()
frame.set_facecolor('0.90')
plt.show()
```



- Different learning rates give different costs and thus different prediction results
- It the learning rate is too large (0.01), the cost may oscillate up and down. It may even diverge (though in this example, using 0.001 still eventually ends up at a good value for the cost).
- A lower cost does not mean a better model. You have to check is there is possibly overfitting. It happens when the training accuracy is a lot higher than the test accuracy.

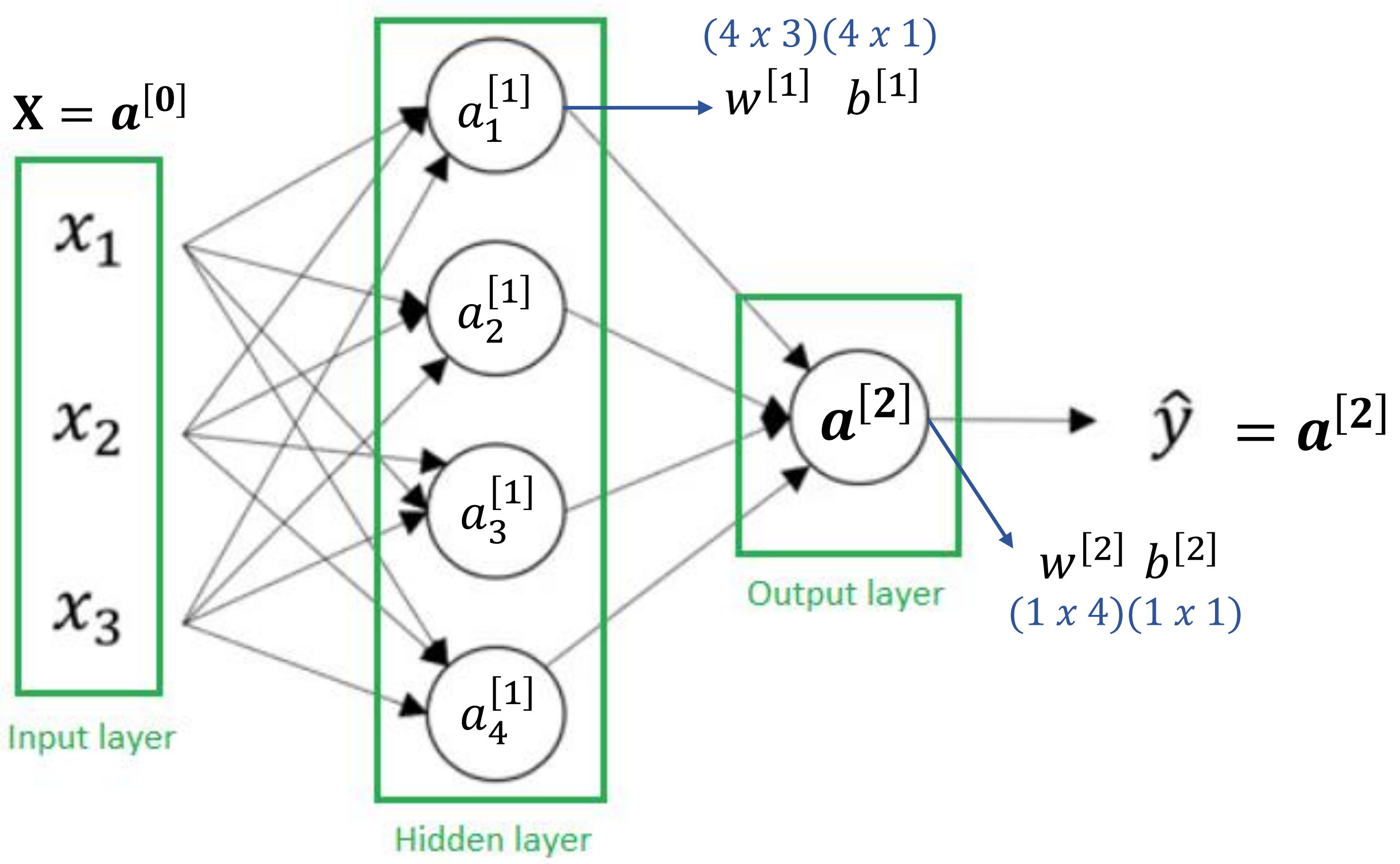
In deep learning we usually recommend that you:

- Choose the learning rate that better minimizes the cost function
- If your model overfits, user other techniques to reduce overfitting.

Week 3- Shallow Neural Networks

The goal is to learn to build a neural network with one hidden layer, using forward propagation and backpropagation.

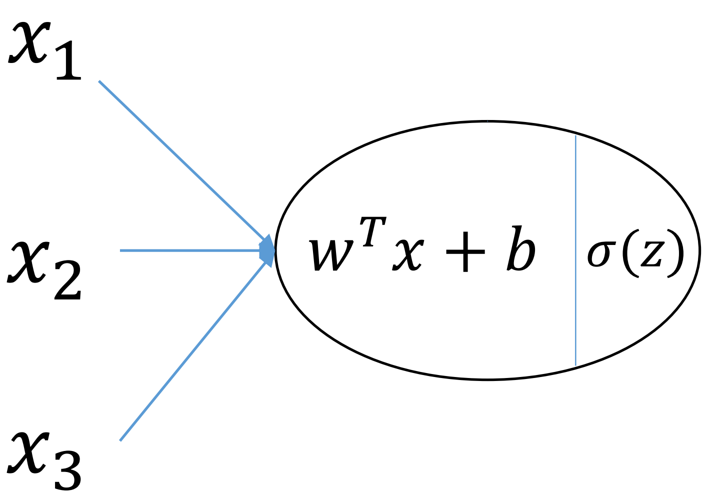
3.1. Neural Network Representation



$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

Fig 4. 2-Layer neural network *(we do not count the input layer)*

3.2. Computing a Neural Network's Output



$$w^T x + b = z$$
$$\sigma(z) = a$$

For $a_1^{[1]}$:

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$$
$$a_1^{[1]} = \sigma(z_1^{[1]})$$

For $a_2^{[1]}$:

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}$$
$$a_2^{[1]} = \sigma(z_2^{[1]})$$

$$Z^{[1]} = \underbrace{\begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ w_3^{[1]T} \\ w_4^{[1]T} \end{bmatrix}}_{}^{\textbf{(4 x 3) } \boldsymbol{w^{[1]}}} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \overset{\textbf{(4 x 1) } \boldsymbol{b^{[1]}}}{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(Z^{[1]})$$

Neural Network Representation learning

Given input x:

$$\underset{(4\ x\ 1)}{z^{[1]}} = \underset{(4\ x\ 3)}{w^{[1]}}\underset{(3\ x\ 1)}{x} + \underset{(4\ x\ 1)}{b^{[1]}} = w^{[1]}a^{[0]}$$

$$\underset{(4\ x\ 1)}{a^{[1]}} = \sigma(\underset{(4\ x\ 1)}{z^{[1]}})$$

$$\underset{(1\ x\ 1)}{z^{[2]}} = \underset{(1\ x\ 4)}{w^{[2]}}\underset{(4\ x\ 1)}{a^{[1]}} + \underset{(1\ x\ 1)}{b^{[2]}}$$

$$\underset{(1\ x\ 1)}{a^{[2]}} = \sigma(\underset{(1\ x\ 1)}{z^{[2]}})$$

3.3. Vectorising across multiple examples

$\boldsymbol{a^{[2](i)}}$: $\boldsymbol{i^{th} example\ of\ layer\ 2}$

$$x \rightarrow a^{[2]} = \hat{y}$$

$$x^{(1)} \rightarrow a^{[2](1)} = \hat{y}^{(1)}$$

$$x^{(2)} \rightarrow a^{[2](2)} = \hat{y}^{(1)}$$

$$\vdots$$

$$x^{(\mathrm{m})} \rightarrow a^{[2](\mathrm{m})} = \hat{y}^{(\mathrm{m})}$$

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = \sigma(Z^{[2]})$$

$$Z^{[1](1)} = W^{[1]}x^{(1)} + \cancelto{0}{b^{[1]}} \qquad Z^{[1](2)} = W^{[1]}x^{(2)} + \cancelto{0}{b^{[1]}} \qquad Z^{[1](3)} = W^{[1]}x^{(3)} + \cancelto{0}{b^{[1]}}$$

$$W^{[1]}x^{(1)} = \begin{bmatrix}\vdots\\ \vdots\end{bmatrix} \qquad W^{[1]}x^{(1)} = \begin{bmatrix}\vdots\\ \vdots\end{bmatrix} \qquad W^{[1]}x^{(1)} = \begin{bmatrix}\vdots\\ \vdots\end{bmatrix}$$

$$\begin{bmatrix} | & | & | \\ X^{(1)} & X^{(1)} & X^{(1)} \\ | & | & | \end{bmatrix} W^{[1]} = \begin{bmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{bmatrix} = \begin{bmatrix} | & | & | \\ Z^{[1](1)} & Z^{[1](2)} & Z^{[1](3)} \\ | & | & | \end{bmatrix} = Z^{[1]}$$

3.4. Activation Functions

The activation function may be different for different layers.

If your output is 0 or 1 (binary classification), the sigmoid function is the desired activation function.

For other case the Rectified Linear Unit (ReLU) is the default activation function, especially when you do not know what to use. And your neural network will run faster.

Leaky ReLU: a = max(0.01z, z) is better than ReLU

Why do you need non-linear activation function?

If we use linear activation function:

$$g(z) = z$$

It is a linear activation function only used in machine learning on the regression problem.

$$a^{[1]} = Z^{[1]} = w^{[1]}x + b^{[1]}$$

$$\begin{aligned} a^{[2]} = Z^{[2]} &= w^{[2]}a^{[1]} + b^{[2]} \\ &= w^{[2]}\left(w^{[1]}x + b^{[1]}\right) \\ &= (w^{[1]}w^{[2]})x + w^{[2]}b^{[1]} + b^{[2]} \\ &= w'x + b' \end{aligned}$$

3.5. Derivatives of Activation Functions

When you implement backpropagation for your neural network, you need to either compute the slope or the derivative of the activation function.

Sigmoid activation function

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

$$\frac{d}{dz}g(z) = slope\ of\ \ g(x)at\ z$$

$$g'(z) = \frac{1}{1+\ e^{-z}}\left(1\ -\frac{1}{1+\ e^{-z}}\right) = g(z)\big(1\ - g(z)\big) = a(1\ - a)$$

$$z = 10\ \rightarrow g(z) \approx 1 \qquad z = -10\ \rightarrow g(z) \approx 0$$

$$g'(z) \approx 1(1-1) \approx 0 \qquad g'(z) \approx 0(1-0) \approx 0$$

Tanh activation function

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

$$\frac{d}{dz} g(z) = slope\ of\ g(x) at\ z = 1 - (\tanh(z))^2 = 1 - a^2$$

$z = 10 \ \rightarrow g(z) \approx 1$

$g'(z) \approx 0$

$z = -10 \ \rightarrow g(z) \approx 1$

$g'(z) \approx 0$

ReLU and Leaky ReLU activation function

$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 0\ if\ z < 0 \\ 1\ if\ z > 0 \\ undefined\ if\ z = 0 \end{cases}$$

$$g(z) = \max(0.01z, z)$$

$$g'(z) = \begin{cases} 0.01\ if\ z < 0 \\ 1\ if\ z \geq 0 \end{cases}$$

3.6. Gradient Descent for Neural Network

$n^x = n^{[0]}$
$n^{[1]}$
$n^{[2]} = 1$

Parameters $\underset{(n^{[1]}, n^{[0]})}{w^{[1]}}$ $\underset{(n^{[1]}, 1)}{b^{[1]}}$ $\underset{(n^{[2]}, n^{[1]})}{w^{[2]}}$ $\underset{(n^{[2]}, 1)}{b^{[2]}}$

Cost Function $$J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

Gradient Descent

$Repeat\{$

$Compute\ predictions\ (\hat{y}^{(i)}, y = 1, 2, \dots, m)$

$$dw^{[1]} = \frac{dJ}{dw^{[1]}}, db^{[1]} = \frac{dJ}{db^{[1]}}, \dots\dots$$

$$w^{[1]} = w^{[1]} - \alpha\, dw^{[1]}$$
$$b^{[1]} = b^{[1]} - \alpha\, db^{[1]}$$
$$w^{[2]} = w^{[2]} - \alpha\, dw^{[2]}$$
$$b^{[2]} = b^{[2]} - \alpha\, db^{[2]}$$

$\}$

Forward Propagation

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g^{[2]}(Z^{[2]})$$
$$= \sigma(Z^{[2]})$$

Backward Propagation

$$dZ^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m} dZ^{[1]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} \mathrm{np.sum}(\mathrm{d}Z^{[2]}, axis = 1, keepdims = True)$$

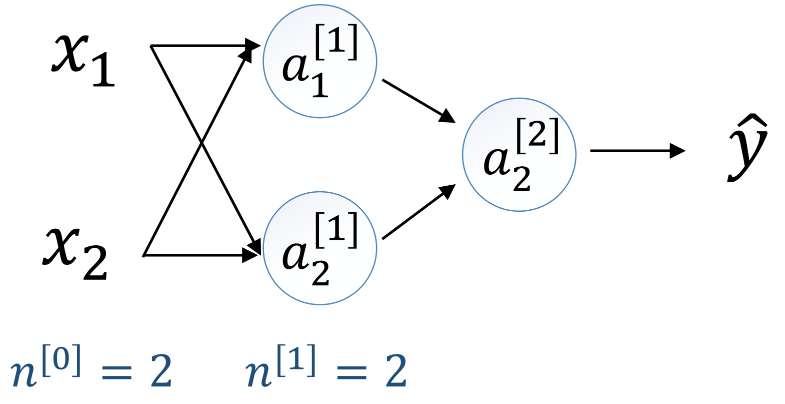
$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * \ g^{[1]'}(Z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$$

$$db^{[1]} = \frac{1}{m} \mathrm{np.sum}(\mathrm{d}Z^{[1]}, axis = 1, keepdims = True)$$

3.7. Random Initialization

We cannot initialize the parameters with zeros.
What happens if you initialize weights with zeros?



$$w^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\downarrow$$

$$\boldsymbol{a_1^{[1]} = a_2^{[1]}} \qquad \boldsymbol{dz_1^{[1]} = dz_2^{[1]}}$$

Random Initialization

$$w^{[1]} = np.random.randn\big((2,1)\big) * 0.01 \longrightarrow \textit{small to speed the gradient descent}$$

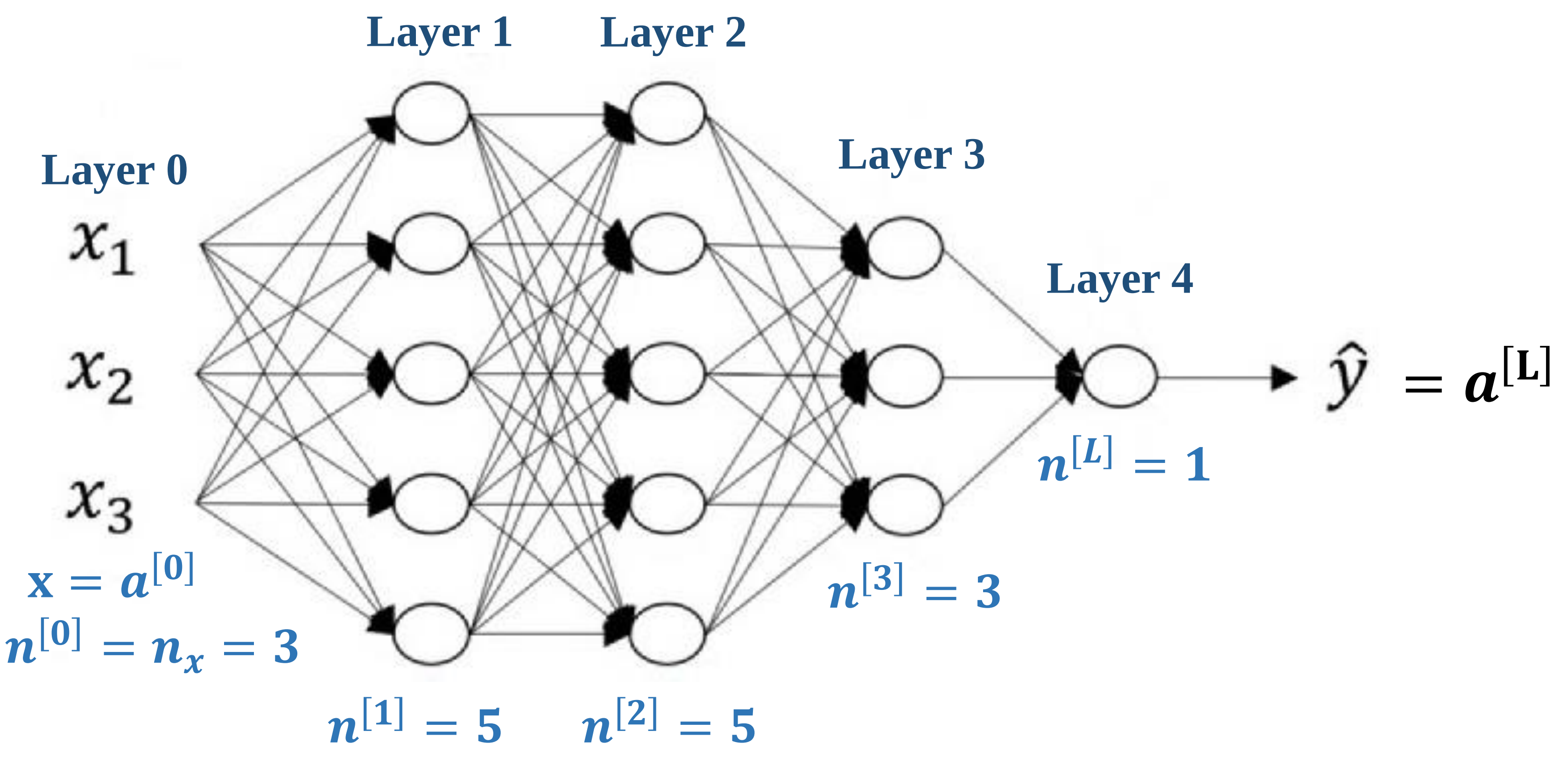
$$b^{[1]} = np.zeros\big((2,1)\big)$$

$$w^{[2]} = np.random.randn\big((2,1)\big) * 0.01$$

Week 4- Deep Neural Networks

The goal is to understand the key computations underlying deep learning, use them to build and train deep neural networks, and apply it to computer vision.

4.1. Deep L-layer neural network



$L = 4$ (# of layers)

$n^{[l]} =$ of units in layer l

$a^{[l]} =$ activation in layer l

$a^{[l]} = g^{[l]}(Z^{[l]})$

4.2. Forward Propagation in a Deep Network

$$z^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$

$$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = g^{[2]}\left(Z^{[2]}\right)$$

$$\dots\dots\dots\dots$$

$$z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$$

$$a^{[4]} = g^{[4]}\left(Z^{[4]}\right) = \hat{y}$$

$$\boldsymbol{z}^{[\mathbf{l}]} = \boldsymbol{w}^{[\mathbf{l}]}\boldsymbol{a}^{[\boldsymbol{l}-\mathbf{1}]} + \boldsymbol{b}^{[\mathbf{l}]}$$

$$\boldsymbol{a}^{[\mathbf{l}]} = \boldsymbol{g}^{[\boldsymbol{l}]}\left(\boldsymbol{z}^{[\mathbf{l}]}\right) \quad (n^{[l]}, 1)$$

Vectorized

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g^{[2]}(Z^{[2]})$$

$$\hat{y} = g(Z^{[4]})$$

$$\boldsymbol{Z}^{[\mathbf{l}]} = \boldsymbol{W}^{[\mathbf{l}]}\boldsymbol{A}^{[\boldsymbol{l}-\mathbf{1}]} + \boldsymbol{b}^{[\mathbf{l}]}$$

$$\boldsymbol{A}^{[\mathbf{l}]} = \boldsymbol{g}^{[\boldsymbol{l}]}\left(\boldsymbol{Z}^{[\mathbf{l}]}\right)$$

4.3. Getting your matrix dimensions right

Parameters $W^{[l]}$ and $b^{[l]}$

$L = 5, n^{[1]} = 3, n^{[2]} = 5, n^{[3]} = 4, n^{[4]} = 2, n^{[3]} = 1$

$$\underset{\substack{(3\,,1)\\(n^{[l]},1)}}{z^{[1]}} = \underset{\substack{(3\,,2)\\(n^{[1]},n^{[0]})}}{w^{[1]}}\underset{\substack{(2\,,1)\\(n^{[0]},1)}}{x} + \underset{(3\,,1)}{b^{[1]}}$$

$\boldsymbol{W}^{[\mathbf{l}]}$: $(n^{[l]}, n^{[l-1]})$ $\boldsymbol{dW}^{[\mathbf{l}]}$: $(n^{[l]}, n^{[l-1]})$

$\boldsymbol{b}^{[\mathbf{l}]}$: $(n^{[l]}, 1)$ $\boldsymbol{db}^{[\mathbf{l}]}$: $(n^{[l]}, 1)$

Vectorized Implementation

$$\underset{(n^{[l]},m)}{Z^{[1]}} = \underset{(n^{[1]},n^{[0]})}{W^{[1]}}\underset{(n^{[0]},m)}{X} + \underset{(n^{[1]},m)}{b^{[1]}}$$

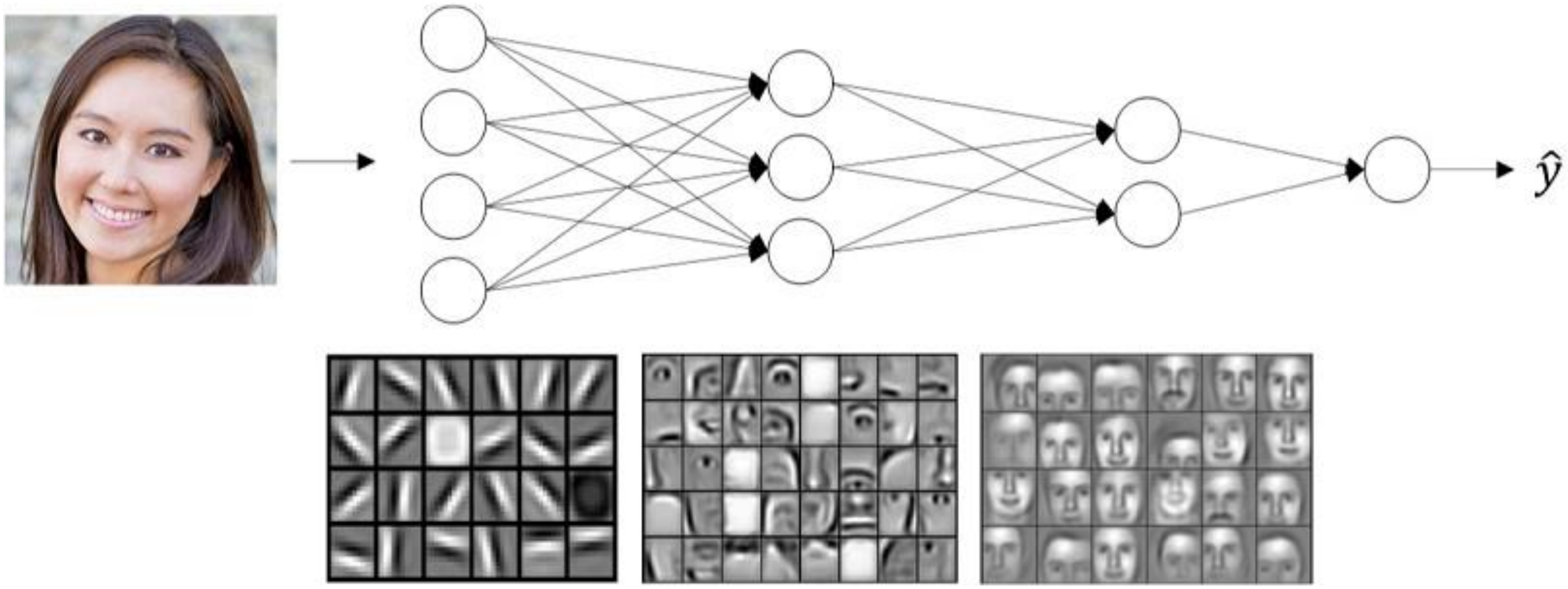
$\boldsymbol{Z}^{[\mathbf{l}]}, \boldsymbol{a}^{[l]}$: $(n^{[l]}, 1)$

$\boldsymbol{Z}^{[\mathbf{l}]}, \boldsymbol{A}^{[l]}$: $(n^{[l]}, m)$

$\boldsymbol{dZ}^{[\mathbf{l}]}, \boldsymbol{dA}^{[l]}$: $(n^{[l]}, m)$

4.4. Why deep representations

Intuition about deep representation



Intuition about deep representation

If you are building a system for face recognition here is what a deep neural network could be doing:

a) Input a face (picture)
b) 1st layer: feature detector or edge detector. Where are the edges in this picture?
c) 2sd layer: detect the edges and group edges together to form part of faces
d) 3rd layer: detect different type of faces

Some people think to make an analogy between deep neural network and the human brain, where we believe, or neuroscientists believe that the human brain also starts detecting simple things like edges in what your eyes see then build up those up to detect more complex things like the faces that you see.

Circuit Theory and Deep Learning

Informally, there are functions you can compute with a small L-layer deep neural network that shallower networks require exponentially more hidden units to compute.

4.5. Building blocks of deep neural networks

Forward and backward propagation functions

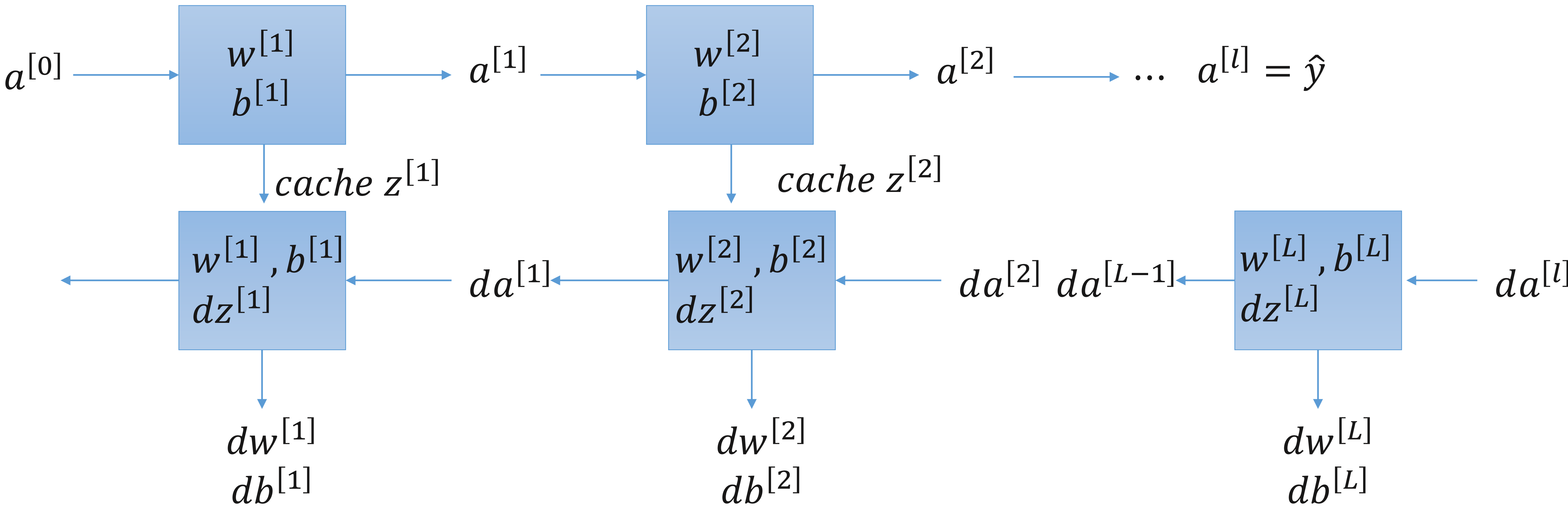
$Layer\ l{:}W^{[l]}, b^{[l]}$

Forward $Input \rightarrow a^{[l-1]}, Output \rightarrow a^{[l]}$

Backward $Input \rightarrow da^{[l]}, cache(z^{[l]}), Output \rightarrow da^{[l-1]}, dw^{[l]}, db^{[l]}$

$\boldsymbol{layer\ l}$

$$\begin{tikzcd}
a^{[l-1]} \arrow[r] & \boxed{\begin{matrix} w^{[l]} \\ b^{[l]} \end{matrix}} \arrow[r] \arrow[d, "cache\ z^{[l]}"] & a^{[l]} \\
da^{[l-1]} & \boxed{\begin{matrix} w^{[l]}, b^{[l]} \\ dz^{[l]} \end{matrix}} \arrow[l] \arrow[d] & da^{[l]} \arrow[l] \\
 & \begin{matrix} dw^{[l]} \\ db^{[l]} \end{matrix} &
\end{tikzcd}$$

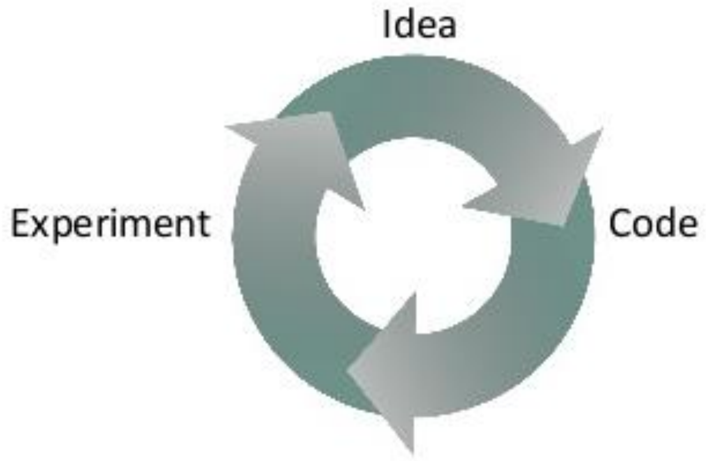


$$w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha\, db^{[l]}$$

4.6. Parameters vs. Hyper-parameters

Parameters: $W^{[l]}$, $b^{[l]}$, ...

Hyper-Parameters: learning rate, # iterations, # hidden layer, # hidden units, activation function



Hyper-parameters control parameters and determine their final value. When we start a project it is very difficult to know the values of the hyper-parametrs, so the only way it to try different values and go around the cycle.

Backward Propagation

$$dZ^{[\mathrm{L}]} = A^{[\mathrm{L}]} - Y$$

$$dW^{[\mathrm{L}]} = \frac{1}{m} dZ^{[L]} A^{[L-1]T}$$

$$db^{[\mathrm{L}]} = \frac{1}{m} \mathrm{np.sum}(\mathrm{d}Z^{[L]}, axis = 1, keepdims = True)$$

$$dZ^{[\mathrm{L}-1]} = W^{[\mathrm{L}]T} dZ^{[L]} * \ g^{[L-1]'}(Z^{[L-1]})$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * \ g^{[1]'}(Z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$$

$$db^{[1]} = \frac{1}{m} \mathrm{np.sum}(\mathrm{d}Z^{[1]}, axis = 1, keepdims = True)$$