INTERNSHIP REPORT

Name: Sayantan Roy
Roll Number: GCECTB-R18-3025
Subject: Neural Networks and Deep Learning
Dept: CSE
Year:4th
Sem:7th

INTERNSHIP REPORT

DeepLearning.AI

COURSE CERTIFICATE

Aug 31, 2021

SAYANTAN ROY

has successfully completed

Neural Networks and Deep Learning

an online non-credit course authorized by DeepLearning.AI and offered through Coursera

Andrew Ng, Founder, DeepLearning.AI & Co-founder, Coursera
Kian Katanforoosh, Co-founder, Workera
Younes Bensouda Mourri, Instructor of AI, Stanford University

Verify at coursera.org/verify/LUMTS7L3F9DC

Coursera has confirmed the identity of this individual and their participation in the course.

ACKNOWLEDGEMENT

First and foremost, I would like to thank our **head of the Computer Science and Engineering Department Mrs. Kalpana Saha(Roy)** who instructed us to get an internship.

I would like to thank **Coursera** and **Deep Learning AI** team who provided us an amazing 4 weeks course on **Neural Networks and Deep Learning** which trained me to become stronger In **deep learning** domain.

I would like to thank our Teacher **Dr. Partha Ghosh** who guided me in doing these internship. He provided us with invaluable advice and helped us in difficult periods. His motivation and help contributed tremendously to the successful completion of the internship and internship report .

Besides, I would like to thank all the teachers who helped me by giving me advice and providing the equipment which I needed.

Also I would like to thank my family and friends for their support. Without that support I couldn't have succeeded in completing this project.

At last but not in least, I would like to thank everyone who helped and motivated me to work on this project.

- Sayantan Roy
GCECTB-R18-3025

LEARNING PATH OF NEURAL NETWORKS AND DEEP LEARNING COURSE(Duration : 4 Weeks)(COURSERA)

Week 1: Introduction to deep learning and neural networks and various learning types : supervised and unsupervised

Week 2 : Numpy basics , activation function and logistic regression, L1 and L2 regularization

Week 3: Building the deep neural network

Week 4: Image classification with neural network

Course Link:https://www.coursera.org/learn/neural-networks-deep-learning

Certificate Link:https://coursera.org/share/1e80971273b618d99957b7054b4d7c36

Github:https://github.com/sayantanr/Neural-Networks-and-Deep-Learning-Coursera-Sayantan-Roy-2021.git

In coursera we learn through video courses, quiz assignments and programming assignments.

So I divided my report into 2 parts. The 1st part will contain the quiz assignments done by me and then the programming assignments.

PART 1 : QUIZ ASSIGNMENT

Congratulations! You passed!

Next Item

1. What does the analogy "AI is the new electricity" refer to?

1 / 1 points

- AI runs on computers and is thus powered by electricity, but it is letting computers do things not possible before.
- Similar to electricity starting about 100 years ago, AI is transforming multiple industries. ✓

Correct
Yes. AI is transforming many fields from the car industry to agriculture to supply-chain...

- AI is powering personal devices in our homes and offices, similar to electricity.
- Through the "smart grid", AI is delivering a new wave of electricity.

2. Which of these are reasons for Deep Learning recently taking off? (Check the three options that apply.)

1 / 1 points

- Deep learning has resulted in significant improvements in important applications such as online advertising, speech recognition, and image recognition. ✓

Correct
These were all examples discussed in lecture 3.

- Neural Networks are a brand new field.

Un-selected is correct

- We have access to a lot more data. ✓

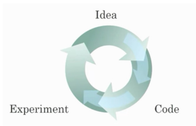
Correct
Yes! The digitalization of our society has played a huge role in this.

- We have access to a lot more computational power. ✓

Correct
Yes! The development of hardware, perhaps especially GPU computing, has significantly improved deep learning algorithms' performance.

3. Recall this diagram of iterating over different ML ideas. Which of the statements below are true? (Check all that apply.)

1 / 1 points



- Being able to try out ideas quickly allows deep learning engineers to iterate more quickly. ✓

Correct
Yes, as discussed in Lecture 4.

- Faster computation can help speed up how long a team takes to iterate to a good idea. ✓

Correct
Yes, as discussed in Lecture 4.

- It is faster to train on a big dataset than a small dataset.

Un-selected is correct

- Recent progress in deep learning algorithms has allowed us to train good models faster (even without changing the CPU/GPU hardware). ✓

Correct
Yes. For example, we discussed how switching from sigmoid to ReLU activation functions allows faster training.

4. When an experienced deep learning engineer works on a new problem, they can usually use insight from previous problems to train a good model on the first try, without needing to iterate multiple times through different models. True/False?

1 / 1 points

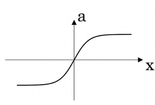
- True
- False ✓

Correct
Yes. Finding the characteristics of a model is key to have good performance. Although experience can help, it requires multiple iterations to build a good model.

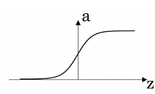
5. Which one of these plots represents a ReLU activation function?

1 / 1 points

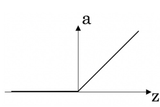
- Figure 1:



- Figure 2:

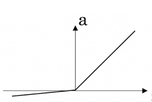


- Figure 3: ✓



Correct
Correct! This is the ReLU activation function, the most used in neural networks.

- Figure 4:



6. Images for cat recognition is an example of "structured" data, because it is represented as a structured array in a computer. True/False?

1 / 1 points

- True
- False ✓

Correct
Yes. Images for cat recognition is an example of "unstructured" data.

7. A demographic dataset with statistics on different cities' population, GDP per capita, economic growth is an example of "unstructured" data because it contains data coming from different sources. True/False?

1 / 1 points

- True
- False ✓

Correct
A demographic dataset with statistics on different cities' population, GDP per capita, economic growth is an example of "structured" data by opposition to image, audio or text datasets.

8. Why is an RNN (Recurrent Neural Network) used for machine translation, say translating English to French? (Check all that apply.)

2 / 2 points

- It can be trained as a supervised learning problem. ✓

Correct
Yes. We can train it on many pairs of sentences x (English) and y (French).

- It is strictly more powerful than a Convolutional Neural Network (CNN).

Un-selected is correct

- It is applicable when the input/output is a sequence (e.g., a sequence of words). ✓

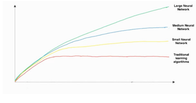
Correct
Yes. An RNN can map from a sequence of english words to a sequence of french words.

- RNNs represent the recurrent process of Idea->Code->Experiment->Idea->....

Un-selected is correct

9. In this diagram which we hand-drew in lecture, what do the horizontal axis (x-axis) and vertical axis (y-axis) represent?

1 / 1 points



- x-axis is the performance of the algorithm
  y-axis (vertical axis) is the amount of data.
- x-axis is the amount of data
  y-axis is the size of the model you train.
- x-axis is the input to the algorithm
  y-axis is outputs.
- x-axis is the amount of data ✓
  y-axis (vertical axis) is the performance of the algorithm.

Correct

10. Assuming the trends described in the previous question's figure are accurate (and hoping you got the axis labels right), which of the following are true? (Check all that apply.)

1 / 1 points

- Increasing the size of a neural network generally does not hurt an algorithm's performance, and it may help significantly. ✓

Correct
Yes. According to the trends in the figure above, big networks usually perform better than small networks.

- Decreasing the training set size generally does not hurt an algorithm's performance, and it may help significantly.

Un-selected is correct

- Decreasing the size of a neural network generally does not hurt an algorithm's performance, and it may help significantly.

Un-selected is correct

- Increasing the training set size generally does not hurt an algorithm's performance, and it may help significantly. ✓

Correct
Yes. Bringing more data to a model is almost always beneficial.

Congratulations! You passed! Next Item

1 / 1 points

1. What does a neuron compute?

- A neuron computes a function g that scales the input x linearly (Wx + b)
- A neuron computes an activation function followed by a linear function (z = Wx + b)
- ✓ A neuron computes a linear function (z = Wx + b) followed by an activation function

Correct
Correct, we generally say that the output of a neuron is a = g(Wx + b) where g is the activation function (sigmoid, tanh, ReLU, ...).

- A neuron computes the mean of all features before applying the output to an activation function

---

1 / 1 points

2. Which of these is the "Logistic Loss"?

- $\mathcal{L}^{(i)}(\hat{y}^{(i)}, y^{(i)}) = \| y^{(i)} - \hat{y}^{(i)} \|$
- $\mathcal{L}^{(i)}(\hat{y}^{(i)}, y^{(i)}) = max(0, y^{(i)} - \hat{y}^{(i)})$
- $\mathcal{L}^{(i)}(\hat{y}^{(i)}, y^{(i)}) = \| y^{(i)} - \hat{y}^{(i)} \|^2$
- ✓ $\mathcal{L}^{(i)}(\hat{y}^{(i)}, y^{(i)}) = -(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}))$

Correct
Correct, this is the logistic loss you've seen in lecture!

---

1 / 1 points

3. Suppose img is a (32,32,3) array, representing a 32x32 image with 3 color channels red, green and blue. How do you reshape this into a column vector?

- ✓ x = img.reshape((32*32*3,1))

Correct

- x = img.reshape((32*32,3))
- x = img.reshape((1,32*32,*3))
- x = img.reshape((3,32*32))

---

1 / 1 points

4. Consider the two following random arrays "a" and "b":

```
a = np.random.randn(2, 3) # a.shape = (2, 3)
b = np.random.randn(2, 1) # b.shape = (2, 1)
c = a + b
```

What will be the shape of "c"?

- c.shape = (3, 2)
- c.shape = (2, 1)
- The computation cannot happen because the sizes don't match. It's going to be "Error"!
- ✓ c.shape = (2, 3)

Correct
Yes! This is broadcasting. b (column vector) is copied 3 times so that it can be summed to each column of a.

---

1 / 1 points

5. Consider the two following random arrays "a" and "b":

```
a = np.random.randn(4, 3) # a.shape = (4, 3)
b = np.random.randn(3, 2) # b.shape = (3, 2)
c = a*b
```

What will be the shape of "c"?

- c.shape = (3, 3)
- c.shape = (4,2)
- ✓ The computation cannot happen because the sizes don't match. It's going to be "Error"!

Correct
Indeed! In numpy the "*" operator indicates element-wise multiplication. It is different from "np.dot()". If you would try "c = np.dot(a,b)" you would get c.shape = (4, 2).

- c.shape = (4, 3)

---

1 / 1 points

6. Suppose you have n_x input features per example. Recall that $X = [x^{(1)} x^{(2)} \dots x^{(m)}]$. What is the dimension of X?

- $(1, m)$
- ✓ (n_x, m)

Correct

- (m, n_x)
- $(m, 1)$

---

1 / 1 points

7. Recall that "np.dot(a,b)" performs a matrix multiplication on a and b, whereas "a*b" performs an element-wise multiplication.

Consider the two following random arrays "a" and "b":

```
a = np.random.randn(12288, 150) # a.shape = (12288, 150)
b = np.random.randn(150, 45) # b.shape = (150, 45)
c = np.dot(a,b)
```

What is the shape of c?

- The computation cannot happen because the sizes don't match. It's going to be "Error"!
- c.shape = (12288, 150)
- c.shape = (150,150)
- ✓ c.shape = (12288, 45)

Correct
Correct, remember that a np.dot(a, b) has shape (number of rows of a, number of columns of b). The sizes match because :

"number of columns of a = 150 = number of rows of b"

---

1 / 1 points

8. Consider the following code snippet:

```
# a.shape = (3,4)
# b.shape = (4,1)

for i in range(3):
  for j in range(4):
    c[i][j] = a[i][j] + b[j]
```

How do you vectorize this?

- c = a.T + b.T
- c = a + b
- ✓ c = a + b.T

Correct

- c = a.T + b

---

1 / 1 points

9. Consider the following code:

```
a = np.random.randn(3, 3)
b = np.random.randn(3, 1)
c = a*b
```

What will be c? (If you're not sure, feel free to run this in python to find out).

- ✓ This will invoke broadcasting, so b is copied three times to become (3,3), and * is an element-wise product so c.shape will be (3, 3)

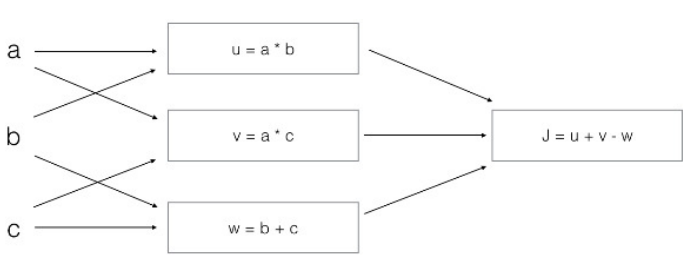
Correct

- This will invoke broadcasting, so b is copied three times to become (3, 3), and * invokes a matrix multiplication operation of two 3x3 matrices so c.shape will be (3, 3)
- This will multiply a 3x3 matrix a with a 3x1 vector, thus resulting in a 3x1 vector. That is, c.shape = (3,1).
- It will lead to an error since you cannot use "*" to operate on these two matrices. You need to instead use np.dot(a,b)

---

1 / 1 points

10. Consider the following computation graph.



What is the output J?

- J = (c - 1)*(b + a)
- ✓ J = (a - 1) * (b + c)

Correct
Yes. J = u + v - w = a*b + a*c - (b + c) = a * (b + c) - (b + c) = (a - 1) * (b + c).

- J = a*b + b*c + a*c
- J = (b - 1) * (c + a)

Congratulations! You passed!

Next Item

1. Which of the following are true? (Check all that apply.)

1 / 1 points

- [x] $a^{[2](12)}$ denotes the activation vector of the 2^{nd} layer for the 12^{th} training example.

  Correct

- [x] X is a matrix in which each column is one training example.

  Correct

- [ ] $a^{[2](12)}$ denotes activation vector of the 12^{th} layer on the 2^{nd} training example.

  Un-selected is correct

- [x] $a^{[2]}$ denotes the activation vector of the 2^{nd} layer.

  Correct

- [x] $a_4^{[2]}$ is the activation output by the 4^{th} neuron of the 2^{nd} layer

  Correct

- [ ] X is a matrix in which each row is one training example.

  Un-selected is correct

- [ ] $a_4^{[2]}$ is the activation output of the 2^{nd} layer for the 4^{th} training example

  Un-selected is correct

---

2. The tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer. True/False?

1 / 1 points

- (•) True

  Correct
  Yes. As seen in lecture the output of the tanh is between -1 and 1, it thus centers the data which makes the learning simpler for the next layer.

- ( ) False

---

3. Which of these is a correct vectorized implementation of forward propagation for layer l, where $1 \leq l \leq L$?

1 / 1 points

- ( ) • $Z^{[l]} = W^{[l]}A^{[l]} + b^{[l]}$
  • $A^{[l+1]} = g^{[l]}(Z^{[l]})$
- ( ) • $Z^{[l]} = W^{[l]}A^{[l]} + b^{[l]}$
  • $A^{[l+1]} = g^{[l+1]}(Z^{[l]})$
- ( ) • $Z^{[l]} = W^{[l-1]}A^{[l]} + b^{[l-1]}$
  • $A^{[l]} = g^{[l]}(Z^{[l]})$
- (•) • $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$
  • $A^{[l]} = g^{[l]}(Z^{[l]})$

  Correct

---

4. You are building a binary classifier for recognizing cucumbers (y=1) vs. watermelons (y=0). Which one of these activation functions would you recommend using for the output layer?

1 / 1 points

- ( ) ReLU
- ( ) Leaky ReLU
- (•) sigmoid

  Correct
  Yes. Sigmoid outputs a value between 0 and 1 which makes it a very good choice for binary classification. You can classify as 0 if the output is less than 0.5 and classify as 1 if the output is more than 0.5. It can be done with tanh as well but it is less convenient as the output is between -1 and 1.

- ( ) tanh

---

5. Consider the following code:

1 / 1 points

```
1  A = np.random.randn(4,3)
2  B = np.sum(A, axis = 1, keepdims = True)
```

What will be B.shape? (If you're not sure, feel free to run this in python to find out).

- ( ) (1, 3)
- ( ) (, 3)
- (•) (4, 1)

  Correct
  Yes, we use (keepdims = True) to make sure that A.shape is (4,1) and not (4,). It makes our code more rigorous.

- ( ) (4,)

---

6. Suppose you have built a neural network. You decide to initialize the weights and biases to be zero. Which of the following statements is true?

1 / 1 points

- (•) Each neuron in the first hidden layer will perform the same computation. So even after multiple iterations of gradient descent each neuron in the layer will be computing the same thing as other neurons.

  Correct

- ( ) Each neuron in the first hidden layer will perform the same computation in the first iteration. But after one iteration of gradient descent they will learn to compute different things because we have "broken symmetry".
- ( ) Each neuron in the first hidden layer will compute the same thing, but neurons in different layers will compute different things, thus we have accomplished "symmetry breaking" as described in lecture.
- ( ) The first hidden layer's neurons will perform different computations from each other even in the first iteration; their parameters will thus keep evolving in their own way.

---

7. Logistic regression's weights w should be initialized randomly rather than to all zeros, because if you initialize to all zeros, then logistic regression will fail to learn a useful decision boundary because it will fail to "break symmetry", True/False?

1 / 1 points

- ( ) True
- (•) False

  Correct
  Yes, Logistic Regression doesn't have a hidden layer. If you initialize the weights to zeros, the first example x fed in the logistic regression will output zero but the derivatives of the Logistic Regression depend on the input x (because there's no hidden layer) which is not zero. So at the second iteration, the weights values follow x's distribution and are different from each other if x is not a constant vector.

---

8. You have built a network using the tanh activation for all the hidden units. You initialize the weights to relative large values, using np.random.randn(..,..)*1000. What will happen?

1 / 1 points

- ( ) This will cause the inputs of the tanh to also be very large, thus causing gradients to also become large. You therefore have to set α to be very small to prevent divergence; this will slow down learning.
- (•) This will cause the inputs of the tanh to also be very large, thus causing gradients to be close to zero. The optimization algorithm will thus become slow.

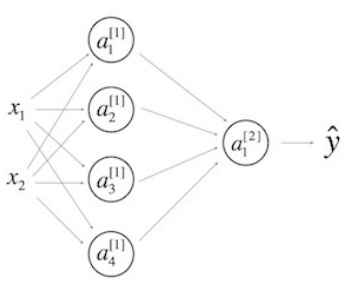
  Correct
  Yes. tanh becomes flat for large values, this leads its gradient to be close to zero. This slows down the optimization algorithm.

- ( ) This will cause the inputs of the tanh to also be very large, causing the units to be "highly activated" and thus speed up learning compared to if the weights had to start from small values.
- ( ) It doesn't matter. So long as you initialize the weights randomly gradient descent is not affected by whether the weights are large or small.

---

9. Consider the following 1 hidden layer neural network:

1 / 1 points



Which of the following statements are True? (Check all that apply).

- [ ] $W^{[1]}$ will have shape (2, 4)

  Un-selected is correct

- [x] $b^{[1]}$ will have shape (4, 1)

  Correct

- [ ] $b^{[1]}$ will have shape (2, 1)

  Un-selected is correct

- [x] $W^{[2]}$ will have shape (1, 4)

  Correct

- [ ] $b^{[2]}$ will have shape (4, 1)

  Un-selected is correct

- [ ] $W^{[2]}$ will have shape (4, 1)

  Un-selected is correct

- [x] $b^{[2]}$ will have shape (1, 1)

  Correct

---

10. In the same network as the previous question, what are the dimensions of $Z^{[1]}$ and $A^{[1]}$?

1 / 1 points

- ( ) $Z^{[1]}$ and $A^{[1]}$ are (1,4)
- (•) $Z^{[1]}$ and $A^{[1]}$ are (4,m)

  Correct

- ( ) $Z^{[1]}$ and $A^{[1]}$ are (4,2)
- ( ) $Z^{[1]}$ and $A^{[1]}$ are (4,1)

Congratulations! You passed!

Next Item

1. What is the "cache" used for in our implementation of forward propagation and backward propagation?

1 / 1 points

- ( ) We use it to pass variables computed during backward propagation to the corresponding forward propagation step. It contains useful values for forward propagation to compute activations.
- ( ) It is used to cache the intermediate values of the cost function during training.
- ( ) It is used to keep track of the hyperparameters that we are searching over, to speed up computation.
- (x) We use it to pass variables computed during forward propagation to the corresponding backward propagation step. It contains useful values for backward propagation to compute derivatives.

Correct
Correct, the "cache" records values from the forward propagation units and sends it to the backward propagation units because it is needed to compute the chain rule derivatives.

2. Among the following, which ones are "hyperparameters"? (Check all that apply.)

1 / 1 points

- [x] number of iterations

Correct

- [ ] bias vectors $b^{[l]}$

Un-selected is correct

- [ ] weight matrices $W^{[l]}$

Un-selected is correct

- [x] number of layers L in the neural network

Correct

- [ ] activation values $a^{[l]}$

Un-selected is correct

- [x] learning rate α

Correct

- [x] size of the hidden layers $n^{[l]}$

Correct

3. Which of the following statements is true?

1 / 1 points

- (x) The deeper layers of a neural network are typically computing more complex features of the input than the earlier layers.

Correct

- ( ) The earlier layers of a neural network are typically computing more complex features of the input than the deeper layers.

4. Vectorization allows you to compute forward propagation in an L-layer neural network without an explicit for-loop (or any other explicit iterative loop) over the layers l=1, 2, …,L. True/False?

1 / 1 points

- ( ) True
- (x) False

Correct
Forward propagation propagates the input through the layers, although for shallow networks we may just write all the lines ($a^{[2]} = g^{[2]}(z^{[2]})$, $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$, ...) in a deeper network, we cannot avoid a for loop iterating over the layers: ($a^{[l]} = g^{[l]}(z^{[l]})$, $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$, ...).

5. Assume we store the values for $n^{[l]}$ in an array called layers, as follows: layer_dims = [n_x, 4,3,2,1]. So layer 1 has four hidden units, layer 2 has 3 hidden units and so on. Which of the following for-loops will allow you to initialize the parameters for the model?

1 / 1 points

( )
```
for(i in range(1, len(layer_dims)/2)):
  parameter['W' + str(i)] = np.random.randn(layers[i], layers[i-1])) *
     0.01
  parameter['b' + str(i)] = np.random.randn(layers[i], 1) * 0.01
```

( )
```
for(i in range(1, len(layer_dims)/2)):
  parameter['W' + str(i)] = np.random.randn(layers[i], layers[i-1])) *
     0.01
  parameter['b' + str(i)] = np.random.randn(layers[i-1], 1) * 0.01
```

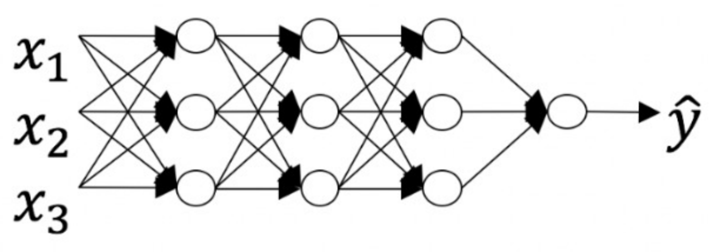
( )
```
for(i in range(1, len(layer_dims))):
  parameter['W' + str(i)] = np.random.randn(layers[i-1], layers[i])) *
     0.01
  parameter['b' + str(i)] = np.random.randn(layers[i], 1) * 0.01
```

(x)
```
for(i in range(1, len(layer_dims))):
  parameter['W' + str(i)] = np.random.randn(layers[i], layers[i-1])) *
     0.01
  parameter['b' + str(i)] = np.random.randn(layers[i], 1) * 0.01
```

Correct

6. Consider the following neural network.

1 / 1 points



How many layers does this network have?

- (x) The number of layers L is 4. The number of hidden layers is 3.

Correct
Yes. As seen in lecture, the number of layers is counted as the number of hidden layers + 1. The input and output layers are not counted as hidden layers.

- ( ) The number of layers L is 3. The number of hidden layers is 3.
- ( ) The number of layers L is 4. The number of hidden layers is 4.
- ( ) The number of layers L is 5. The number of hidden layers is 4.

7. During forward propagation, in the forward function for a layer l you need to know what is the activation function in a layer (Sigmoid, tanh, ReLU, etc.). During backpropagation, the corresponding backward function also needs to know what is the activation function for layer l, since the gradient depends on it. True/False?

1 / 1 points

- (x) True

Correct
Yes, as you've seen in the week 3 each activation has a different derivative. Thus, during backpropagation you need to know which activation was used in the forward propagation to be able to compute the correct derivative.

- ( ) False

8. There are certain functions with the following properties:

1 / 1 points

(i) To compute the function using a shallow network circuit, you will need a large network (where we measure size by the number of logic gates in the network), but (ii) To compute it using a deep network circuit, you need only an exponentially smaller network. True/False?

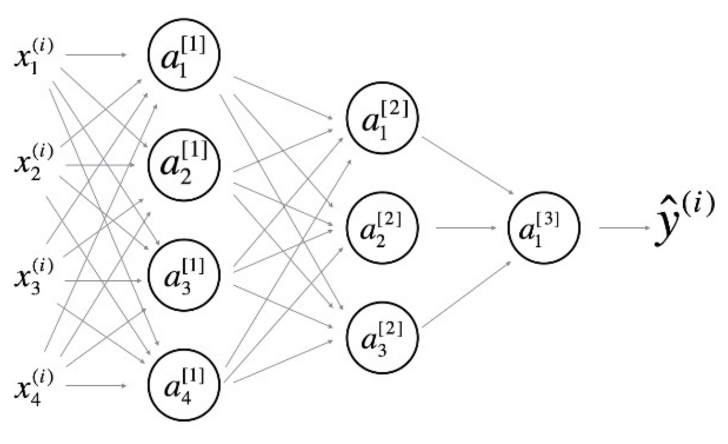
- (x) True

Correct

- ( ) False

9. Consider the following 2 hidden layer neural network:

1 / 1 points



Which of the following statements are True? (Check all that apply).

- [x] $W^{[1]}$ will have shape (4, 4)

Correct
Yes. More generally, the shape of $W^{[l]}$ is $(n^{[l]}, n^{[l-1]})$.

- [x] $b^{[1]}$ will have shape (4, 1)

Correct
Yes. More generally, the shape of $b^{[l]}$ is $(n^{[l]}, 1)$.

- [ ] $W^{[1]}$ will have shape (3, 4)

Un-selected is correct

- [ ] $b^{[1]}$ will have shape (3, 1)

Un-selected is correct

- [x] $W^{[2]}$ will have shape (3, 4)

Correct
Yes. More generally, the shape of $W^{[l]}$ is $(n^{[l]}, n^{[l-1]})$.

- [ ] $b^{[2]}$ will have shape (1, 1)

Un-selected is correct

- [ ] $W^{[2]}$ will have shape (3, 1)

Un-selected is correct

- [x] $b^{[2]}$ will have shape (3, 1)

Correct
Yes. More generally, the shape of $b^{[l]}$ is $(n^{[l]}, 1)$.

- [ ] $W^{[3]}$ will have shape (3, 1)

Un-selected is correct

- [x] $b^{[3]}$ will have shape (1, 1)

Correct
Yes. More generally, the shape of $b^{[l]}$ is $(n^{[l]}, 1)$.

- [x] $W^{[3]}$ will have shape (1, 3)

Correct
Yes. More generally, the shape of $W^{[l]}$ is $(n^{[l]}, n^{[l-1]})$.

- [ ] $b^{[3]}$ will have shape (3, 1)

Un-selected is correct

10. Whereas the previous question used a specific network, in the general case what is the dimension of W^{[l]}, the weight matrix associated with layer l?

1 / 1 points

- (x) $W^{[l]}$ has shape $(n^{[l]}, n^{[l-1]})$

Correct
True

- ( ) $W^{[l]}$ has shape $(n^{[l-1]}, n^{[l]})$
- ( ) $W^{[l]}$ has shape $(n^{[l]}, n^{[l+1]})$
- ( ) $W^{[l]}$ has shape $(n^{[l+1]}, n^{[l]})$

PART 2: PROGRAMMING ASSIGNMENT

1 Neural Networks and Deep Learning

1.1 Python Basics with numpy (optional)

Welcome to your first (Optional) programming exercise of the deep learning specialization. This exercise gives you a brief introduction to Python. Even if you've used Python before, this will help familiarize you with functions we'll need. In this assignment you will:

- Learn how to use numpy.
- Implement some basic core deep learning functions such as the softmax, sigmoid, dsigmoid, etc...
- Learn how to handle data by normalizing inputs and reshaping images.
- Recognize the importance of vectorization.
- Understand how python broadcasting works.

This assignment prepares you well for the upcoming assignment. Take your time to complete it and make sure you get the expected outputs when working through the different exercises. In some code blocks, you will find a "#GRADED FUNCTION: functionName" comment. Please do not modify it. After you are done, submit your work and check your results. You need to score 70% to pass. Good luck :) !

Let's get started!

1.1.1 About iPython Notebooks

iPython Notebooks are interactive coding environments embedded in a webpage. You will be using iPython notebooks in this class. You only need to write code between the ### START CODE HERE ### and ### END CODE HERE ### comments. After writing your code, you can run the cell by either pressing "SHIFT"+"ENTER" or by clicking on "Run Cell" (denoted by a play symbol) in the upper bar of the notebook.

We will often specify "(≈ lines of code)" in the comments to tell you about how much code you need to write. It is just a rough estimate, so don't feel bad if your code is longer or shorter.

Exercise: Set test to "Hello World" in the cell below to print "Hello World" and run the two code below.

```
### START CODE HERE ### ( 1 line of code)
test = "Hello World"
### END CODE HERE ###

print ("test: " + test)

#output
test: Hello World
```

What you need to remember:

- Run your cells using SHIFT+ENTER (or "Run cell")

- Write code in the designated areas using Python 3 only
- Do not modify the code outside of the designated areas

1.1.2 Building basic functions with numpy

Numpy is the main package for scientific computing in Python. It is maintained by a large community (www.numpy.org). In this exercise you will learn several key numpy functions such as np.exp, np.log, and np.reshape. You will need to know how to use these functions for future assignments.

1.1.2.1 sigmoid function, np.exp()

Before using np.exp(), you will use math.exp() to implement the sigmoid function. You will then see why np.exp() is preferable to math.exp().

Exercise: Build a function that returns the sigmoid of a real number x. Use math.exp(x) for the exponential function.

Reminder: $sigmoid(x) = \frac{1}{1+e^{-x}}$ is sometimes also known as the logistic function. It is a non-linear function used not only in Machine Learning (Logistic Regression), but also in Deep Learning.

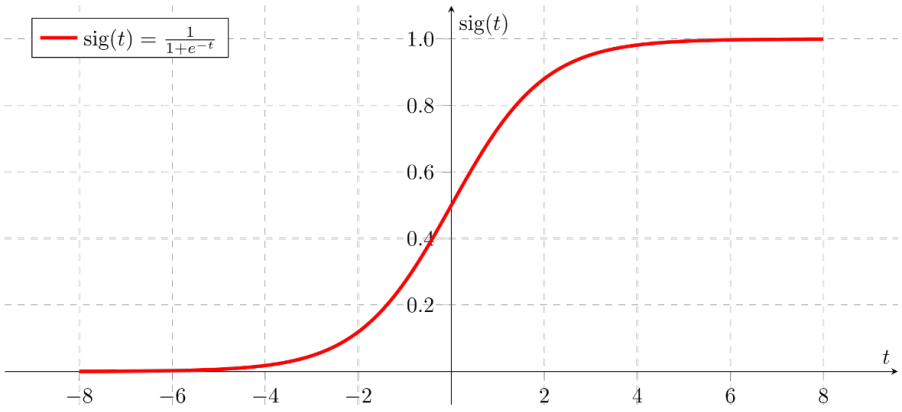


Figure 1.1.1 $sigmoid(t) = \frac{1}{1+e^{-t}}$

To refer to a function belonging to a specific package you could call it using package_name.function(). Run the code below to see an example with math.exp().

```
# GRADED FUNCTION: basic_sigmoid
import math

def basic_sigmoid(x):
    """
    Compute sigmoid of x.

    Arguments:
    x -- A scalar

    Return:
```

```
    s -- sigmoid(x)
    """

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

    return s
```

Actually, we rarely use the "math" library in deep learning because the inputs of the functions are real numbers. In deep learning we mostly use matrices and vectors. This is why numpy is more useful.

In fact, if $x = (x_1, x_2, ..., x_n)$ is a row vector then $np.exp(x)$ will apply the exponential function to every element of x. The output will thus be: $np.exp(x) = (e^{x_1}, e^{x_2}, ..., e^{x_n})$

```
import numpy as np

# example of np.exp
x = np.array([1, 2, 3])
print(np.exp(x)) # result is (exp(1), exp(2), exp(3))

#output
[  2.71828183   7.3890561   20.08553692]
```

Furthermore, if x is a vector, then a Python operation such as $s = x + 3$ or $s = \frac{1}{x}$ will output s as a vector of the same size as x.

```
# example of vector operation
x = np.array([1, 2, 3])
print (x + 3)

#output
[4 5 6]
```

Exercise: Implement the sigmoid function using numpy.

Instructions: x could now be either a real number, a vector, or a matrix. The data structures we use in numpy to represent these shapes (vectors, matrices...) are called numpy arrays. You don't need to know more for now.

For x $\in \mathbb{R}^n$,

$$sigmoid(x) = sigmoid \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix} = \begin{pmatrix} \frac{1}{1+e^{-x_1}} \\ \frac{1}{1+e^{-x_2}} \\ ... \\ \frac{1}{1+e^{-x_n}} \end{pmatrix} \tag{1.1.1}$$

```
# GRADED FUNCTION: sigmoid
import numpy as np # this means you can access numpy functions by
↪ writing np.function() instead of numpy.function()

def sigmoid(x):
```

```
    """
    Compute the sigmoid of x

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

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

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

    return s
```

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

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

1.1.2.2 Sigmoid gradient

As you've seen in lecture, you will need to compute gradients to optimize loss functions using backpropagation. Let's code your first gradient function.

Exercise: Implement the function sigmoid_grad() to compute the gradient of the sigmoid function with respect to its input x. The formula is:

$$sigmoid\_derivative(x) = \sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{1.1.2}$$

You often code this function in two steps:

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

```
# GRADED FUNCTION: sigmoid_derivative
def sigmoid_derivative(x):
    """
    Compute the gradient (also called the slope or derivative) of the
↪   sigmoid function with respect to its input x.
    You can store the output of the sigmoid function into variables and
↪   then use it to calculate the gradient.

    Arguments:
    x -- A scalar or numpy array
```

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

    ### START CODE HERE ### ( 2 lines of code)
    s = 1/(1+np.exp(-x))
    ds = s*(1-s)
    ### END CODE HERE ###

    return ds
```

1.1.2.3 Reshaping arrays

Two common numpy functions used in deep learning are np.shape and np.reshape().

- X.shape is used to get the shape (dimension) of a matrix/vector X.
- X.reshape(...) is used to reshape X into some other dimension.

For example, in computer science, an image is represented by a 3D array of shape $(length, height, depth = 3)$. However, when you read an image as the input of an algorithm you convert it to a vector of shape $(length * height * 3, 1)$. In other words, you "unroll", or reshape, the 3D array into a 1D vector.

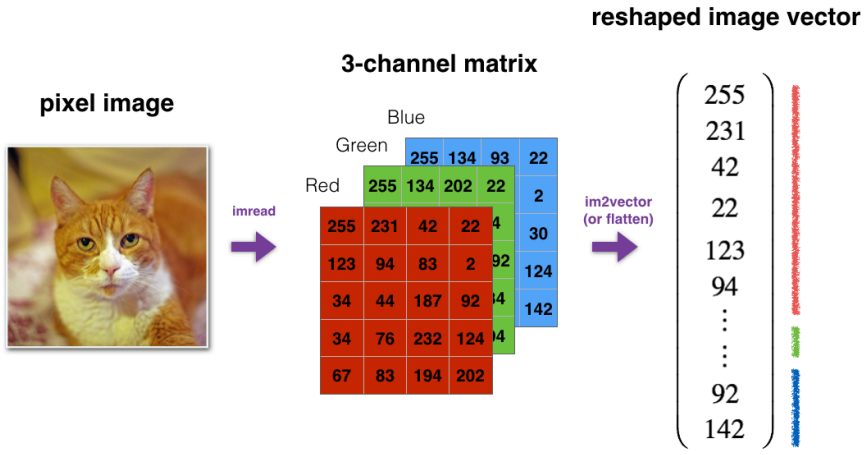


Figure 1.1.2 Reshape image a vector

Exercise: Implement "image2vector()" that takes an input of shape (length, height, 3) and returns a vector of shape (lengthheight3, 1). For example, if you would like to reshape an array v of shape (a, b, c) into a vector of shape (a*b,c) you would do:

```
v = v.reshape((v.shape[0]*v.shape[1], v.shape[2])) # v.shape[0] = a ;
↪  v.shape[1] = b ; v.shape[2] = c
```

Please don't hardcode the dimensions of image as a constant. Instead look up the quantities you need with "image.shape[0]", etc.

```
# GRADED FUNCTION: image2vector
def image2vector(image):
    """
    Argument:
    image -- a numpy array of shape (length, height, depth)

    Returns:
    v -- a vector of shape (length*height*depth, 1)
    """

    ### START CODE HERE ### (  1 line of code)
    v = image.reshape((image.shape[0]*image.shape[1]*image.shape[2]),1)
    ### END CODE HERE ###

    return v
```

1.1.2.4 Normalizing rows

Another common technique we use in Machine Learning and Deep Learning is to normalize our data. It often leads to a better performance because gradient descent converges faster after normalization. Here, by normalization we mean changing x to $\frac{x}{\|x\|}$ (dividing each row vector of x by its norm).

For example, if

$$x = \begin{bmatrix} 0 & 3 & 4 \\ 2 & 6 & 4 \end{bmatrix} \tag{1.1.3}$$

then

$$\|x\| = np.linalg.norm(x, axis = 1, keepdims = True) = \begin{bmatrix} 5 \\ \sqrt{56} \end{bmatrix} \tag{1.1.4}$$

and

$$x\_normalized = \frac{x}{\|x\|} = \begin{bmatrix} 0 & \frac{3}{5} & \frac{4}{5} \\ \frac{2}{\sqrt{56}} & \frac{6}{\sqrt{56}} & \frac{4}{\sqrt{56}} \end{bmatrix} \tag{1.1.5}$$

Note that you can divide matrices of different sizes and it works fine: this is called broadcasting and you're going to learn about it in part 1.1.2.5.

Exercise: Implement normalizeRows() to normalize the rows of a matrix. After applying this function to an input matrix x, each row of x should be a vector of unit length (meaning length 1).

```
# GRADED FUNCTION: normalizeRows
def normalizeRows(x):
    """
    Implement a function that normalizes each row of the matrix x (to
↪   have unit length).

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

    Returns:
```

```
    x -- The normalized (by row) numpy matrix. You are allowed to
↪   modify x.
    """

    ### START CODE HERE ### ( 2 lines of code)
    # Compute x_norm as the norm 2 of x. Use np.linalg.norm(..., ord =
    ↪  2, axis = ..., keepdims = True)
    x_norm = np.linalg.norm(x,axis=1,keepdims=True)

    # Divide x by its norm.
    x = x/x_norm
    ### END CODE HERE ###

    return x
```

```
x = np.array([
    [0, 3, 4],
    [1, 6, 4]])
print("normalizeRows(x) = " + str(normalizeRows(x)))

#output
normalizeRows(x) = [[ 0.          0.6         0.8       ]
 [ 0.13736056  0.82416338  0.54944226]]
```

Note: In normalizeRows(), you can try to print the shapes of x_norm and x, and then rerun the assessment. You'll find out that they have different shapes. This is normal given that x_norm takes the norm of each row of x. So x_norm has the same number of rows but only 1 column. So how did it work when you divided x by x_norm? This is called broadcasting and we'll talk about it now!

1.1.2.5 Broadcasting and the softmax function

A very important concept to understand in numpy is "broadcasting". It is very useful for performing mathematical operations between arrays of different shapes. For the full details on broadcasting, you can read the official broadcasting documentation.

Exercise: Implement a softmax function using numpy. You can think of softmax as a normalizing function used when your algorithm needs to classify two or more classes. You will learn more about softmax in the second course of this specialization.

Instructions:

- for x $\in \mathbb{R}^{1\times n}$,

$$\begin{aligned} softmax(x) &= softmax(\begin{bmatrix} x_1 & x_2 & ... & x_n \end{bmatrix}) \\ &= \begin{bmatrix} \frac{e^{x_1}}{\sum_j e^{x_j}} & \frac{e^{x_2}}{\sum_j e^{x_j}} & ... & \frac{e^{x_n}}{\sum_j e^{x_j}} \end{bmatrix} \end{aligned} \tag{1.1.6}$$

- for a matrix $x \in \mathbb{R}^{m\times n}$, x_{ij} maps to the element in the i^{th} row and j^{th} column of x,

thus we have:

$$
\begin{aligned}
softmax(x) &= softmax \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix} \\
&= \begin{bmatrix} \frac{e^{x_{11}}}{\sum_j e^{x_{1j}}} & \frac{e^{x_{12}}}{\sum_j e^{x_{1j}}} & \frac{e^{x_{13}}}{\sum_j e^{x_{1j}}} & \dots & \frac{e^{x_{1n}}}{\sum_j e^{x_{1j}}} \\ \frac{e^{x_{21}}}{\sum_j e^{x_{2j}}} & \frac{e^{x_{22}}}{\sum_j e^{x_{2j}}} & \frac{e^{x_{23}}}{\sum_j e^{x_{2j}}} & \dots & \frac{e^{x_{2n}}}{\sum_j e^{x_{2j}}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{e^{x_{m1}}}{\sum_j e^{x_{mj}}} & \frac{e^{x_{m2}}}{\sum_j e^{x_{mj}}} & \frac{e^{x_{m3}}}{\sum_j e^{x_{mj}}} & \dots & \frac{e^{x_{mn}}}{\sum_j e^{x_{mj}}} \end{bmatrix} \\
&= \begin{pmatrix} softmax(\text{first row of x}) \\ softmax(\text{second row of x}) \\ \dots \\ softmax(\text{last row of x}) \end{pmatrix}
\end{aligned}
\tag{1.1.7}
$$

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

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

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

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

    # Apply exp() element-wise to x. Use np.exp(...).
    x_exp = np.exp(x)

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

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

    return s
```

Note:

If you print the shapes of x_exp, x_sum and s above and rerun the assessment cell, you will see that x_sum is of shape (2,1) while x_exp and s are of shape (2,5). x_exp/x_sum works due to python broadcasting.

Congratulations! You now have a pretty good understanding of python numpy and have implemented a few useful functions that you will be using in deep learning.

What you need to remember:

- **np.exp(x) works for any np.array x and applies the exponential function to every coordinate**
- **the sigmoid function and its gradient**
- **image2vector is commonly used in deep learning**
- **np.reshape is widely used. In the future, you'll see that keeping your matrix/vector dimensions straight will go toward eliminating a lot of bugs.**
- **numpy has efficient built-in functions**
- **broadcasting is extremely useful**

1.1.3 Vectorization

In deep learning, you deal with very large datasets. Hence, a non-computationally-optimal function can become a huge bottleneck in your algorithm and can result in a model that takes ages to run. To make sure that your code is computationally efficient, you will use vectorization. For example, try to tell the difference between the following implementations of the dot/outer/elementwise product.

```
import time

x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

### CLASSIC DOT PRODUCT OF VECTORS IMPLEMENTATION ###
tic = time.process_time()
dot = 0
for i in range(len(x1)):
    dot+= x1[i]*x2[i]
toc = time.process_time()
print ("dot = " + str(dot) + "\n ----- Computation time = " +
↪  str(1000*(toc - tic)) + "ms")

### CLASSIC OUTER PRODUCT IMPLEMENTATION ###
tic = time.process_time()
outer = np.zeros((len(x1),len(x2))) # we create a len(x1)*len(x2)
↪  matrix with only zeros
for i in range(len(x1)):
    for j in range(len(x2)):
        outer[i,j] = x1[i]*x2[j]
toc = time.process_time()
print ("outer = " + str(outer) + "\n ----- Computation time = " +
↪  str(1000*(toc - tic)) + "ms")
```

```
### CLASSIC ELEMENTWISE IMPLEMENTATION ###
tic = time.process_time()
mul = np.zeros(len(x1))
for i in range(len(x1)):
    mul[i] = x1[i]*x2[i]
toc = time.process_time()
print ("elementwise multiplication = " + str(mul) + "\n -----
↪  Computation time = " + str(1000*(toc - tic)) + "ms")

### CLASSIC GENERAL DOT PRODUCT IMPLEMENTATION ###
W = np.random.rand(3,len(x1)) # Random 3*len(x1) numpy array
tic = time.process_time()
gdot = np.zeros(W.shape[0])
for i in range(W.shape[0]):
    for j in range(len(x1)):
        gdot[i] += W[i,j]*x1[j]
toc = time.process_time()
print ("gdot = " + str(gdot) + "\n ----- Computation time = " +
↪  str(1000*(toc - tic)) + "ms")
```

```
#output
dot = 278
 ----- Computation time = 0.17511900000011238ms
 outer = [[ 81.  18.  18.  81.   0.  81.  18.  45.   0.   0.  81.  18.
 ↪  45.   0.
    0.]
 [ 18.   4.   4.  18.   0.  18.   4.  10.   0.   0.  18.   4.  10.   0.
    0.]
 [ 45.  10.  10.  45.   0.  45.  10.  25.   0.   0.  45.  10.  25.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [ 63.  14.  14.  63.   0.  63.  14.  35.   0.   0.  63.  14.  35.   0.
    0.]
 [ 45.  10.  10.  45.   0.  45.  10.  25.   0.   0.  45.  10.  25.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [ 81.  18.  18.  81.   0.  81.  18.  45.   0.   0.  81.  18.  45.   0.
    0.]
 [ 18.   4.   4.  18.   0.  18.   4.  10.   0.   0.  18.   4.  10.   0.
    0.]
```

```
 [ 45.  10.  10.  45.   0.  45.  10.  25.   0.   0.  45.  10.  25.   0.
   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.]]
 ----- Computation time = 0.3380989999999251ms
elementwise multiplication = [ 81.   4.  10.   0.   0.  63.  10.   0.
 ↪  0.   0.  81.   4.  25.   0.   0.]
 ----- Computation time = 0.1734490000000477ms
gdot = [ 25.22022143  27.33603654  20.18059712]
 ----- Computation time = 0.2346690000001317ms
```

```
x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

### VECTORIZED DOT PRODUCT OF VECTORS ###
tic = time.process_time()
dot = np.dot(x1,x2)
toc = time.process_time()
print ("dot = " + str(dot) + "\n ----- Computation time = " +
 ↪  str(1000*(toc - tic)) + "ms")

### VECTORIZED OUTER PRODUCT ###
tic = time.process_time()
outer = np.outer(x1,x2)
toc = time.process_time()
print ("outer = " + str(outer) + "\n ----- Computation time = " +
 ↪  str(1000*(toc - tic)) + "ms")

### VECTORIZED ELEMENTWISE MULTIPLICATION ###
tic = time.process_time()
mul = np.multiply(x1,x2)
toc = time.process_time()
print ("elementwise multiplication = " + str(mul) + "\n -----
 ↪  Computation time = " + str(1000*(toc - tic)) + "ms")

### VECTORIZED GENERAL DOT PRODUCT ###
tic = time.process_time()
dot = np.dot(W,x1)
toc = time.process_time()
print ("gdot = " + str(dot) + "\n ----- Computation time = " +
 ↪  str(1000*(toc - tic)) + "ms")
```

```
#output
dot = 278
 ----- Computation time = 0.1825449999999229ms
```

```
outer = [[81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [63 14 14 63  0 63 14 35  0  0 63 14 35  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
 ----- Computation time = 0.15074200000020355ms
elementwise multiplication = [81  4 10  0  0 63 10  0  0  0 81  4 25  0
↪  0]
 ----- Computation time = 0.13718599999990033ms
gdot = [ 25.22022143  27.33603654  20.18059712]
 ----- Computation time = 0.4201209999998845ms
```

As you may have noticed, the vectorized implementation is much cleaner and more efficient. For bigger vectors/matrices, the differences in running time become even bigger.

Note that np.dot() performs a matrix-matrix or matrix-vector multiplication. This is different from np.multiply() and the * operator (which is equivalent to .* in Matlab/Octave), which performs an element-wise multiplication.

1.1.3.1 Implement the L1 and L2 loss functions

Exercise: Implement the numpy vectorized version of the L1 loss. You may find the function abs(x) (absolute value of x) useful.

Reminder:

- The loss is used to evaluate the performance of your model. The bigger your loss is, the more different your predictions ($\hat{y}$) are from the true values (y). In deep learning, you use optimization algorithms like Gradient Descent to train your model and to minimize the cost.
- L1 loss is defined as:

$$L_1(\hat{y}, y) = \sum_{i=0}^{m} |y^{(i)} - \hat{y}^{(i)}| \tag{1.1.8}$$

```
# GRADED FUNCTION: L1
def L1(yhat, y):
    """
    Arguments:
    yhat -- vector of size m (predicted labels)
    y -- vector of size m (true labels)
```

```
    Returns:
    loss -- the value of the L1 loss function defined above
    """

    loss = sum(abs(y-yhat))

    return loss
```

```
yhat = np.array([.9, 0.2, 0.1, .4, .9])
y = np.array([1, 0, 0, 1, 1])
print("L1 = " + str(L1(yhat,y)))
```

Exercise: Implement the numpy vectorized version of the L2 loss. There are several way of implementing the L2 loss but you may find the function np.dot() useful. As a reminder, if $x = [x_1, x_2, ..., x_n]$, then "np.dot(x,x)" $= \sum_{j=0}^{n} x_j^2$.

L2 loss is defined as

$$L_2(\hat{y}, y) = \sum_{i=0}^{m} (y^{(i)} - \hat{y}^{(i)})^2 \tag{1.1.9}$$

```
# GRADED FUNCTION: L2
def L2(yhat, y):
    """
    Arguments:
    yhat -- vector of size m (predicted labels)
    y -- vector of size m (true labels)

    Returns:
    loss -- the value of the L2 loss function defined above
    """

    loss = np.dot(y-yhat,y-yhat)

    return loss
```

```
yhat = np.array([.9, 0.2, 0.1, .4, .9])
y = np.array([1, 0, 0, 1, 1])
print("L2 = " + str(L2(yhat,y)))
```

What to remember:

- **Vectorization is very important in deep learning. It provides computational efficiency and clarity.**
- **You have reviewed the L1 and L2 loss.**
- **You are familiar with many numpy functions such as np.sum, np.dot, np.multiply, np.maximum, etc...**

1.2 Logistic Regression with a Neural Network mindset

Welcome to the first (required) programming exercise of the deep learning specialization. In this notebook you will build your first image recognition algorithm. You will build a cat classifier that recognizes cats with 70% accuracy!

Figure 1.2.1 cat

As you keep learning new techniques you will increase it to 80+ % accuracy on cat vs. non-cat datasets. By completing this assignment you will:

- Work with logistic regression in a way that builds intuition relevant to neural networks.
- Learn how to minimize the cost function.
- Understand how derivatives of the cost are used to update parameters.

Take your time to complete this assignment and make sure you get the expected outputs when working through the different exercises. In some code blocks, you will find a "#GRADED FUNCTION: functionName" comment. Please do not modify these comments. After you are done, submit your work and check your results. You need to score 70% to pass. Good luck :) !

Instructions:

- Do not use loops (for/while) in your code, unless the instructions explicitly ask you to do so.

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.

1.2.1 Packages

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

- 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 Python.
- PIL 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

#matplotlib inline
```

1.2.2 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).

You will build a simple image-recognition algorithm that can correctly classify pictures as cat or non-cat.

Let's get more familiar with the dataset. Load the data by running the following code.

```
# Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes =
↪  load_dataset()
```

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 train_set_y and test_set_y don't need any preprocessing).
Each line of your train_set_x_orig and test_set_x_orig is an array representing an image. You can visualize an example by running the following code. Feel free also to change the index value and re-run to see other images.

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

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

Exercise: Find the values for:

- m_train (number of training examples)
- m_test (number of test examples)
- num_px (= height = width of a training image)

Remember that train_set_x_orig is a numpy-array of shape (m_train, num_px, num_px, 3). For instance, you can access m_train by writing train_set_x_orig.shape[0].

```
### START CODE HERE ### ( 3 lines of code)
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]
### END CODE HERE ###

print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/Width of each image: num_px = " + str(num_px))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) +
↪  ", 3)")
print ("train_set_x shape: " + str(train_set_x_orig.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x shape: " + str(test_set_x_orig.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

Output for m_train, m_test and num_px:

```
Number of training examples: m_train = 209
Number of testing examples: m_test = 50
Height/Width of each image: num_px = 64
Each image is of size: (64, 64, 3)
train_set_x shape: (209, 64, 64, 3)
train_set_y shape: (1, 209)
test_set_x shape: (50, 64, 64, 3)
test_set_y shape: (1, 50)
```

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.

Exercise: Reshape the training and test data sets so that images of size (num_px, num_px, 3) are flattened into single vectors of shape (num_px*num_px*3, 1).

A trick when you want to flatten a matrix X of shape (a,b,c,d) to a matrix X_flatten of shape (b * c *d, a) is to use:

```
X_flatten = X.reshape(-1,X.shape[0])
```

```
# Reshape the training and test examples

### START CODE HERE ### ( 2 lines of code)
train_set_x_flatten =
↪  train_set_x_orig.reshape(train_set_x_orig.shape[0],-1).T
test_set_x_flatten =
↪  test_set_x_orig.reshape(test_set_x_orig.shape[0],-1).T
### END CODE HERE ###

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]))
```

Output for train_set_x_flatten shape, test_set_x_flatten shape:

```
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 31 56 22 33]
```

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 substract 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 picture datasets, it is simpler and more convenient and works almost as well to just divide every row of the dataset by 255 (the maximum value of a pixel channel).

Let's standardize our dataset.

```
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.
```

What 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

1.2.3 General Architecture of the learning algorithm

It's time to design a simple algorithm to distinguish cat images from non-cat images.

You will build a Logistic Regression, using a Neural Network mindset. The following Figure explains why **Logistic Regression is actually a very simple Neural Network**!

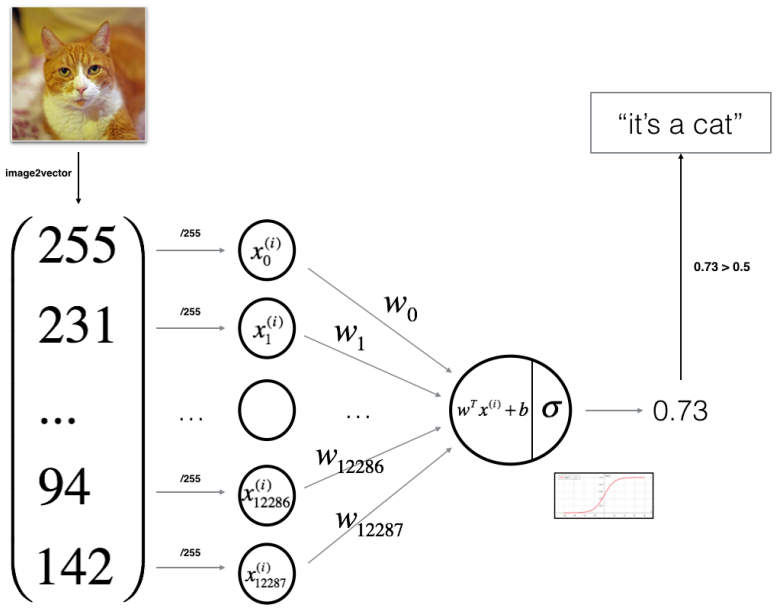


Figure 1.2.2 Principle of Logistic Regression

Mathematical expression of the algorithm:
For one example $x^{(i)}$:

$$z^{(i)} = w^T x^{(i)} + b \tag{1.2.1}$$

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)}) \tag{1.2.2}$$

$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)}) \tag{1.2.3}$$

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

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}) \tag{1.2.4}$$

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 test set)
- Analyse the results and conclude

1.2.4 Building the parts of our algorithm

The main steps for building a Neural Network are:

- Define the model structure (such as number of input features)
- Initialize the model's parameters
- Loop:
  - Calculate current loss (forward propagation)
  - Calculate current gradient (backward propagation)
  - Update parameters (gradient descent)

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

1.2.4.1 Helper functions

Exercise: Using your code from "Python Basics", implement sigmoid(). As you've seen in the figure above, you need to compute $sigmoid(w^T x + b) = \frac{1}{1+e^{-(w^T x+b)}}$ to make predictions. Use np.exp().

```
# GRADED FUNCTION: sigmoid

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

    Arguments:
    z -- A scalar or numpy array of any size.

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

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

    return s
```

1.2.4.2 Initializing parameters

Exercise: Implement parameter initialization in the cell below. You have to initialize w as a vector of zeros. If you don't know what numpy function to use, look up np.zeros() in the Numpy library's documentation.

```python
def initialize_with_zeros(dim):
    """
    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)
    """

    ### START CODE HERE ### ( 1 line of code)
    w = np.zeros((dim,1))
    b = 0
    ### END CODE HERE ###

    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b
```

For image inputs, w will be of shape (num_px × num_px × 3, 1).

1.2.4.3 Forward and Backward propagation

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

Exercise: Implement a function propagate() that computes the cost function and its gradient.

Hints:

Forward Propagation:

- You get X
- You compute $A = \sigma(w^T X + b) = (a^{(0)}, a^{(1)}, ..., a^{(m-1)}, a^{(m)})$
- 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:

$$\frac{\partial J}{\partial w} = \frac{1}{m} X (A - Y)^T \tag{1.2.5}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) \tag{1.2.6}$$

Code is as follows:

```
# GRADED FUNCTION: propagate

def propagate(w, b, X, Y):
    """
    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()
    """

    m = X.shape[1]

    # FORWARD PROPAGATION (FROM X TO COST)
    ### START CODE HERE ### ( 2 lines of code)
    A =  sigmoid(np.dot(w.T,X)+b)    # compute activation
    cost =  -(np.dot(Y,np.log(A.T))+np.dot(np.log(1-A),(1-Y).T))/m  #
     ↪  compute cost
    ### END CODE HERE ###

    # BACKWARD PROPAGATION (TO FIND GRAD)
    ### START CODE HERE ### ( 2 lines of code)
    dw = np.dot(X,(A-Y).T)/m
    db = np.sum(A-Y)/m
    ### END CODE HERE ###

    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())

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

    return grads, cost
```

1.2.4.4 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.

Exercise: Write down the optimization function. The goal is to learn w and b by minimizing the cost function J. For a parameter θ, the update rule is $\theta = \theta - \alpha\ d\theta$, where α is the learning rate.

```
# GRADED FUNCTION: optimize

def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost =
↪ False):
    """
    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.
    """

    costs = []

    for i in range(num_iterations):


        # Cost and gradient calculation ( 1-4 lines of code)
        ### START CODE HERE ###
        grads, cost = propagate(w, b, X, Y)
```

```
        ### END CODE HERE ###

        # Retrieve derivatives from grads
        dw = grads["dw"]
        db = grads["db"]

        # update rule ( 2 lines of code)
        ### START CODE HERE ###
        w = w-learning_rate*dw
        b = b-learning_rate*db
        ### END CODE HERE ###

        # Record the costs
        if i % 100 == 0:
            costs.append(cost)

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

    params = {"w": w,
              "b": b}

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

    return params, grads, costs
```

```
w, b, X, Y = np.array([[1.],[2.]]), 2.,
↪  np.array([[1.,2.,-1.],[3.,4.,-3.2]]), np.array([[1,0,1]])
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"]))

#output
w = [[ 0.19033591]
 [ 0.12259159]]
b = 1.92535983008
dw = [[ 0.67752042]
 [ 1.41625495]]
db = 0.219194504541
```

Exercise: The previous function will output the learned w and b. We are able to use w and b to predict the labels for a dataset X. Implement the predict() function. There is two steps to computing predictions:

- Calculate $\hat{Y} = A = \sigma(w^T X + b)$
- Convert the entries of a into 0 (if activation <= 0.5) or 1 (if activation > 0.5), stores the predictions in a vector 'Y_prediction'. If you wish, you can use an 'if'/'else' statement in a 'for' loop (though there is also a way to vectorize this).

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

    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)

    Returns:
    Y_prediction -- a numpy array (vector) containing all predictions
 ↪  (0/1) for the examples in 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 in the picture
    ### START CODE HERE ### ( 1 line of code)
    A = sigmoid(np.dot(w.T,X)+b)
    ### END CODE HERE ###

    for i in range(A.shape[1]):

        # Convert probabilities A[0,i] to actual predictions p[0,i]
        ### START CODE HERE ### ( 4 lines of code)
        if A[0][i]<=0.5:A[0][i]=0
        else: A[0][i]=1
    Y_prediction=A
        ### END CODE HERE ###

    assert(Y_prediction.shape == (1, m))

    return Y_prediction
```

What to remember: You've 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**

1.2.5 Merge all functions into a model

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

Exercise: Implement the model function. Use the following notation:

- Y_prediction for your predictions on the test set
- Y_prediction_train for your predictions on the train set
- w, costs, grads for the outputs of optimize()

Code is as follows:

```
# GRADED FUNCTION: model

def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000,
↪  learning_rate = 0.5, print_cost = False):
    """
    Builds the logistic regression model by calling the function you've
↪  implemented previously

    Arguments:
    X_train -- training set represented by a numpy array of shape
↪  (num_px * num_px * 3, m_train)
    Y_train -- training labels represented by a numpy array (vector) of
↪  shape (1, m_train)
    X_test -- test set represented by a numpy array of shape (num_px *
↪  num_px * 3, m_test)
    Y_test -- test labels represented by a numpy array (vector) of
↪  shape (1, m_test)
    num_iterations -- hyperparameter representing the number of
↪  iterations to optimize the parameters
    learning_rate -- hyperparameter representing the learning rate used
↪  in the update rule of optimize()
    print_cost -- Set to true to print the cost every 100 iterations

    Returns:
    d -- dictionary containing information about the model.
    """

    ### START CODE HERE ###

    # initialize parameters with zeros ( 1 line of code)
    w, b = initialize_with_zeros(X_train.shape[0])

    # Gradient descent ( 1 line of code)
    parameters, grads, costs = optimize(w, b, X_train, Y_train,
    ↪  num_iterations, learning_rate, print_cost)

    # Retrieve parameters w and b from dictionary "parameters"
```

```
    w = parameters["w"]
    b = parameters["b"]

    # Predict test/train set examples ( 2 lines of code)
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)

    ### END CODE HERE ###

    # 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))


    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
```

Run the following cell to train your model.

```
d = model(train_set_x, train_set_y, test_set_x, test_set_y,
↪  num_iterations = 2000, learning_rate = 0.005, print_cost = True)

#Output:

Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584508
Cost after iteration 200: 0.466949
Cost after iteration 300: 0.376007
Cost after iteration 400: 0.331463
Cost after iteration 500: 0.303273
Cost after iteration 600: 0.279880
Cost after iteration 700: 0.260042
Cost after iteration 800: 0.242941
Cost after iteration 900: 0.228004
Cost after iteration 1000: 0.214820
Cost after iteration 1100: 0.203078
Cost after iteration 1200: 0.192544
Cost after iteration 1300: 0.183033
Cost after iteration 1400: 0.174399
Cost after iteration 1500: 0.166521
```

```
Cost after iteration 1600: 0.159305
Cost after iteration 1700: 0.152667
Cost after iteration 1800: 0.146542
Cost after iteration 1900: 0.140872
train accuracy: 99.04306220095694 %
test accuracy: 70.0 %
```

Comment: Training accuracy is close to 100%. This is a good sanity check: your model is working and has high enough capacity to fit the training data. Test error is 68%. It is actually not bad for this simple model, given the small dataset we used and that logistic regression is a linear classifier. But no worries, you'll build an even better classifier next week!

Also, you see that the model is clearly overfitting the training data. Later in this specialization you will learn how to reduce overfitting, for example by using regularization. Using the code below (and changing the index variable) you can look at predictions on pictures of the test set.

Let's also plot the cost function and the gradients.

```python
# 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()
```

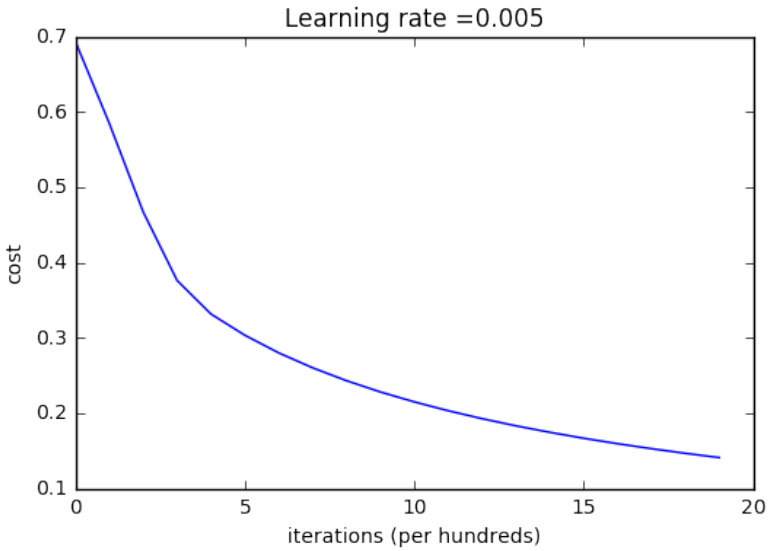


Figure 1.2.3 cost function

Interpretation: 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. Try to increase the number of iterations in the cell above and rerun the cells. You might see that the training set accuracy goes up, but the test set accuracy goes down. This is called **overfitting.**

```
d = model(train_set_x, train_set_y, test_set_x, test_set_y,
↪  num_iterations = 2500, learning_rate = 0.005, print_cost = True)

#Output:

train accuracy: 99.52153110047847 %
test accuracy: 68.0 %
```

1.2.6 Further analysis (optional/ungraded exercise)

Congratulations on building your first image classification model. Let's analyze it further, and examine possible choices for the learning rate α.

Choice of learning rate

Reminder: In order for Gradient Descent to work you must choose the learning rate wisely. The learning rate α determines how rapidly we update the parameters. If the learning rate is too large we may "overshoot" the optimal value. Similarly, if it is too small we will need too many iterations to converge to the best values. That's why it is crucial to use a well-tuned learning rate.

Let's compare the learning curve of our model with several choices of learning rates. Run the cell below. This should take about 1 minute. Feel free also to try different values than the three we have initialized the learning_rates variable to contain, and see what happens.

```
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')

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

The result:

```
learning rate is: 0.01
train accuracy: 99.52153110047847 %
test accuracy: 68.0 %

-------------------------------------

learning rate is: 0.001
train accuracy: 88.99521531100478 %
test accuracy: 64.0 %

-------------------------------------

learning rate is: 0.0001
train accuracy: 68.42105263157895 %
test accuracy: 36.0 %

-------------------------------------
```

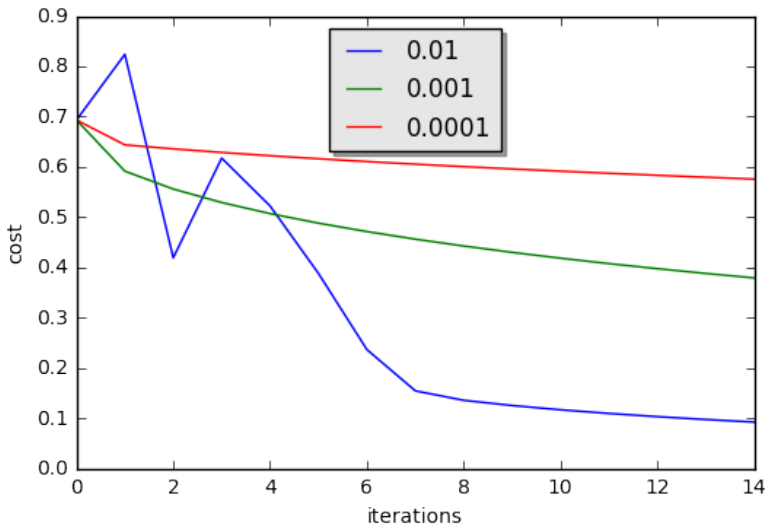


Figure 1.2.4 compare the learning curve of model with three learning rates

Interpretation:

- Different learning rates give different costs and thus different predictions results.
- If the learning rate is too large (0.01), the cost may oscillate up and down. It may even diverge (though in this example, using 0.01 still eventually ends up at a good value for the cost).
- A lower cost doesn't mean a better model. You have to check if 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, use other techniques to reduce overfitting. (We'll talk about this in later videos.)

1.2.7 Test with your own image (optional/ungraded exercise)

Congratulations on finishing this assignment. You can use your own image and see the output of your model. To do that:

1. Click on "File" in the upper bar of this notebook, then click "Open" to go on your Coursera Hub.
2. Add your image to this Jupyter Notebook's directory, in the "images" folder
3. Change your image's name in the following code
4. Run the code and check if the algorithm is right (1 = cat, 0 = non-cat)!

```
my_image = "test2.jpg"   # change this to the name of your image file

# We preprocess the image to fit your algorithm.
fname = "images/" + my_image
image = np.array(ndimage.imread(fname, flatten=False))
my_image = scipy.misc.imresize(image, size=(num_px,num_px)).reshape((1,
 ↪  num_px*num_px*3)).T
my_predicted_image = predict(d["w"], d["b"], my_image)

plt.imshow(image)
print("y = " + str(np.squeeze(my_predicted_image)) + ", your algorithm
 ↪  predicts a \"" +
 ↪  classes[int(np.squeeze(my_predicted_image)),].decode("utf-8") +  "\"
 ↪  picture.")
```

What to remember from this assignment:

1. Preprocessing the dataset is important.
2. You implemented each function separately: initialize(), propagate(), optimize(). Then you built a model().
3. Tuning the learning rate (which is an example of a "hyperparameter") can make a big difference to the algorithm. You will see more examples of this later in this course!

1.2.8 Code of Logistic Regression with a Neural Network

```
#Logistic Regression with a Neural Network mindset

# -*- coding: utf-8 -*-

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

# Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes =
↪  load_dataset()


m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]

# Reshape the training and test examples
train_set_x_flatten =
↪  train_set_x_orig.reshape(train_set_x_orig.shape[0],-1).T
test_set_x_flatten =
↪  test_set_x_orig.reshape(test_set_x_orig.shape[0],-1).T

#"Standardize" the data
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.


# GRADED FUNCTION: sigmoid
def sigmoid(x):
    """
    Compute the sigmoid of x

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

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

    s = 1/(1+np.exp(-x))

    return s
```

```
# GRADED FUNCTION: initialize_with_zeros
def initialize_with_zeros(dim):
    """
    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)
    """

    w = np.zeros((dim,1))
    b = 0

    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b


# GRADED FUNCTION: propagate
def propagate(w, b, X, Y):
    """
    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()
    """
```

```
    m = X.shape[1]

    # FORWARD PROPAGATION (FROM X TO COST)
    A =  sigmoid(np.dot(w.T,X)+b)    # compute activation
    cost =  -(np.dot(Y,np.log(A.T))+np.dot(np.log(1-A),(1-Y).T))/m  #
    ↪  compute cost

    # BACKWARD PROPAGATION (TO FIND GRAD)
    dw = np.dot(X,(A-Y).T)/m
    db = np.sum(A-Y)/m

    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())

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

    return grads, cost

# GRADED FUNCTION: optimize
def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost =
↪  False):
    """
    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.
    """

    costs = []

    for i in range(num_iterations):


        # Cost and gradient calculation ( 1-4 lines of code)
        grads, cost = propagate(w, b, X, Y)

        # Retrieve 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 examples
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    params = {"w": w,
              "b": b}

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

    return params, grads, costs


# GRADED FUNCTION: predict
def predict(w, b, X):
    '''
    Predict whether the label is 0 or 1 using learned logistic
↪   regression parameters (w, b)

    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)

    Returns:
    Y_prediction -- a numpy array (vector) containing all predictions
↪   (0/1) for the examples in 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 in the picture
    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]
        if A[0][i]<=0.5:A[0][i]=0
        else: A[0][i]=1
    Y_prediction=A

    assert(Y_prediction.shape == (1, m))

    return Y_prediction

#=============================================================================
# Merge all functions into a model
#=============================================================================

def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000,
↪   learning_rate = 0.5, print_cost = False):
    """
    Builds the logistic regression model by calling the function you've
↪   implemented previously

    Arguments:
    X_train -- training set represented by a numpy array of shape
↪   (num_px * num_px * 3, m_train)
    Y_train -- training labels represented by a numpy array (vector) of
↪   shape (1, m_train)
    X_test -- test set represented by a numpy array of shape (num_px *
↪   num_px * 3, m_test)
    Y_test -- test labels represented by a numpy array (vector) of
↪   shape (1, m_test)
    num_iterations -- hyperparameter representing the number of
↪   iterations to optimize the parameters
    learning_rate -- hyperparameter representing the learning rate used
↪   in the update rule of optimize()
```

```
    print_cost -- Set to true to print the cost every 100 iterations

    Returns:
    d -- dictionary containing information about the model.
    """

    # initialize parameters with zeros ( 1 line of code)
    w, b =  initialize_with_zeros(X_train.shape[0])

    # Gradient descent ( 1 line of code)
    parameters, grads, costs = optimize(w, b, X_train, Y_train,
    ↪  num_iterations, learning_rate, print_cost)

    # Retrieve parameters w and b from dictionary "parameters"
    w = parameters["w"]
    b = parameters["b"]

    # Predict test/train set examples ( 2 lines of code)
    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))


    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)
```

1.3 Planar data classification with a hidden layer

Welcome to the second programming exercise of the deep learning specialization. In this notebook you will generate red and blue points to form a flower. You will then fit a neural network to correctly classify the points. You will try different layers and see the results.

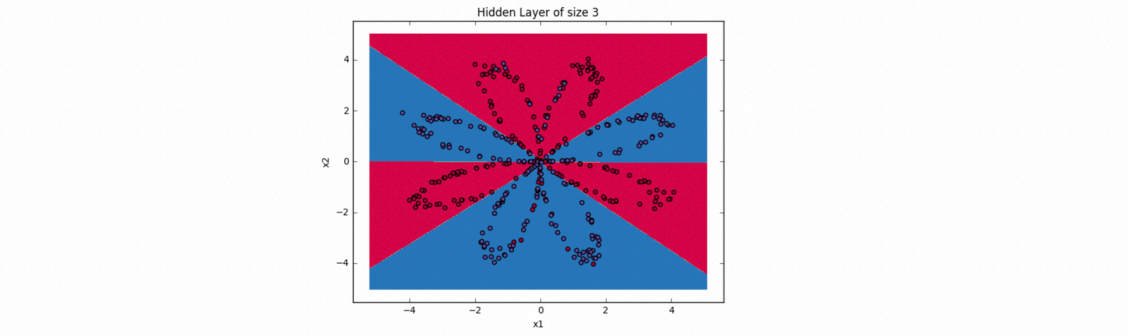


Figure 1.3.1 classify points

By completing this assignment you will:

- Develop an intuition of back-propagation and see it work on data.
- Recognize that the more hidden layers you have the more complex structure you could capture.
- Build all the helper functions to implement a full model with one hidden layer.

You will learn how to:

- Implement a 2-class classification neural network with a single hidden layer
- Use units with a non-linear activation function, such as tanh
- Compute the cross entropy loss
- Implement forward and backward propagation

This assignment prepares you well for the upcoming assignment. Take your time to complete it and make sure you get the expected outputs when working through the different exercises. In some code blocks, you will find a "#GRADED FUNCTION: functionName" comment. Please do not modify it. After you are done, submit your work and check your results. You need to score 70% to pass. Good luck :) !

1.3.1 Packages

Let's first import all the packages that you will need during this assignment.

- numpy is the fundamental package for scientific computing with Python.
- sklearn provides simple and efficient tools for data mining and data analysis.
- matplotlib is a library for plotting graphs in Python.

- testCases provides some test examples to assess the correctness of your functions
- planar_utils provide various useful functions used in this assignment

```
# Package imports
import numpy as np
import matplotlib.pyplot as plt
from testCases_v2 import *
import sklearn
import sklearn.datasets
import sklearn.linear_model
from planar_utils import plot_decision_boundary, sigmoid,
↪  load_planar_dataset, load_extra_datasets

#matplotlib inline

np.random.seed(1) # set a seed so that the results are consistent
```

1.3.2 Dataset

First, let's get the dataset you will work on. The following code will load a "flower" 2-class dataset into variables X and Y.

```
X, Y = load_planar_dataset()
```

Visualize the dataset using matplotlib. The data looks like a "flower" with some red (label y=0) and some blue (y=1) points. Your goal is to build a model to fit this data.

```
# Visualize the data:
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```

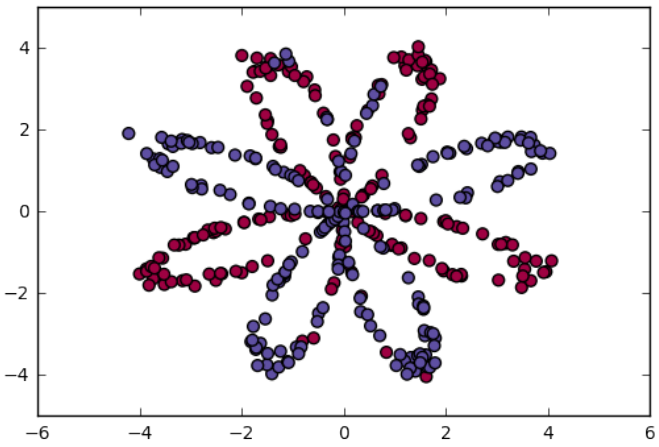


Figure 1.3.2 Visualize the dataset

You have:

- a numpy-array (matrix) X that contains your features (x1, x2)
- a numpy-array (vector) Y that contains your labels (red:0, blue:1).

Lets first get a better sense of what our data is like.

Exercise: How many training examples do you have? In addition, what is the shape of the variables X and Y?

```
### START CODE HERE ### ( 3 lines of code)
shape_X = X.shape
shape_Y = Y.shape
m = shape_X[1]  # training et size
### END CODE HERE ###

print ('The shape of X is: ' + str(shape_X))
print ('The shape of Y is: ' + str(shape_Y))
print ('I have m = %d training examples.' % (m))

#output
The shape of X is: (2, 400)
The shape of Y is: (1, 400)
I have m = 400 training examples.
```

1.3.3 Simple Logistic Regression

Before building a full neural network, lets first see how logistic regression performs on this problem. You can use sklearn's built-in functions to do that. Run the code below to train a logistic regression classifier on the dataset.

```
# Train the logistic regression classifier
clf = sklearn.linear_model.LogisticRegressionCV();
clf.fit(X.T, Y.T);
```

You can now plot the decision boundary of these models. Run the code below

```
# Plot the decision boundary for logistic regression
plot_decision_boundary(lambda x: clf.predict(x), X, Y)
plt.title("Logistic Regression")

# Print accuracy
LR_predictions = clf.predict(X.T)
print ('Accuracy of logistic regression: %d ' %
↪  float((np.dot(Y,LR_predictions) +
↪  np.dot(1-Y,1-LR_predictions))/float(Y.size)*100) +
       '% ' + "(percentage of correctly labelled datapoints)")
```

The result is as follows:

```
Accuracy of logistic regression: 47 % (percentage of correctly labelled
↪  datapoints)
```

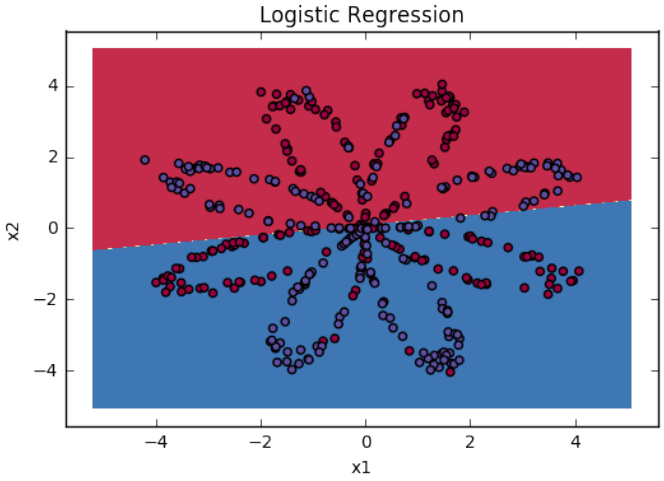


Figure 1.3.3 Logistic Regression

Interpretation: The dataset is not linearly separable, so logistic regression doesn't perform well. Hopefully a neural network will do better. Let's try this now!

1.3.4 Neural Network model

Logistic regression did not work well on the "flower dataset". You are going to train a Neural Network with a single hidden layer.

Here is our model:

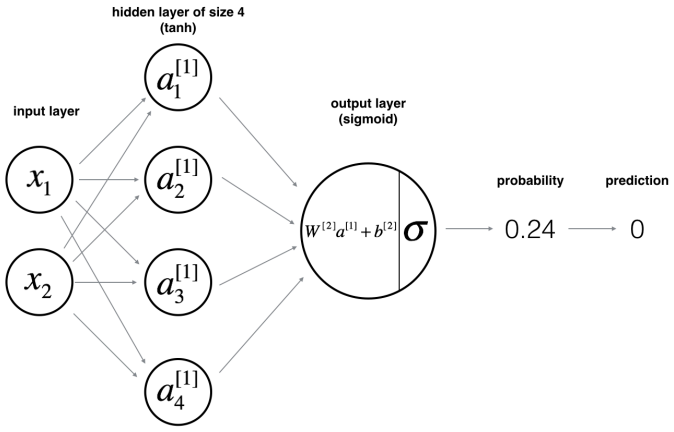


Figure 1.3.4 Neural Network Model

Mathematically:

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

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1](i)} \tag{1.3.1}$$

$$a^{[1](i)} = \tanh(z^{[1](i)}) \tag{1.3.2}$$

$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2](i)} \tag{1.3.3}$$

$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)}) \tag{1.3.4}$$

$$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{1.3.5}$$

Given the predictions on all the examples, you can also compute the cost J as follows:

$$J = -\frac{1}{m}\sum_{i=0}^{m}\left(y^{(i)}\log\left(a^{[2](i)}\right) + (1-y^{(i)})\log\left(1-a^{[2](i)}\right)\right) \tag{1.3.6}$$

Reminder: 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
- Compute loss
- Implement backward propagation to get the gradients
- Update parameters (gradient descent)

You often build helper functions to compute steps 1-3 and then merge them into one function we call 'nn_model()'. Once you've built 'nn_model()' and learnt the right parameters, you can make predictions on new data.

1.3.4.1 Defining the neural network structure

Exercise: Define three variables:

- n_x: the size of the input layer
- n_h: the size of the hidden layer (set this to 4)
- n_y: the size of the output layer

Hint: Use shapes of X and Y to find n_x and n_y. Also, hard code the hidden layer size to be 4.

```
# GRADED FUNCTION: layer_sizes

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_h -- the size of the hidden layer
    n_y -- the size of the output layer
    """
    ### START CODE HERE ### ( 3 lines of code)
    n_x = X.shape[0] # size of input layer
    n_h = 4
    n_y = Y.shape[0]# size of output layer
    ### END CODE HERE ###
    return (n_x, n_h, n_y)
```

1.3.4.2 Initialize the model's parameters

Exercise: Implement the function initialize_parameters().
Instructions:

- Make sure your parameters' sizes are right. Refer to the neural network figure above if needed.
- You will initialize the weights matrices with random values.
  - Use: np.random.randn(a,b) * 0.01 to randomly initialize a matrix of shape (a,b).
- You will initialize the bias vectors as zeros.
  - Use: np.zeros((a,b)) to initialize a matrix of shape (a,b) with zeros.

```
# GRADED FUNCTION: initialize_parameters

def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer

    Returns:
    params -- python dictionary containing your parameters:
                    W1 -- weight matrix of shape (n_h, n_x)
                    b1 -- bias vector of shape (n_h, 1)
                    W2 -- weight matrix of shape (n_y, n_h)
                    b2 -- bias vector of shape (n_y, 1)
    """

    np.random.seed(2) # we set up a seed so that your output matches
    ↪  ours although the initialization is random.
```

```
    ### START CODE HERE ### ( 4 lines of code)
    W1 = np.random.randn(n_h,n_x) * 0.01
    b1 = np.zeros((n_h,1))
    W2 = np.random.randn(n_y,n_h)* 0.01
    b2 = np.zeros((n_y,1))
    ### END CODE HERE ###

    assert (W1.shape == (n_h, n_x))
    assert (b1.shape == (n_h, 1))
    assert (W2.shape == (n_y, n_h))
    assert (b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

1.3.4.3 The Loop

Question: Implement 'forward_propagation()'.
Instructions:

- Look above at the mathematical representation of your classifier.
- You can use the function 'sigmoid()'. It is built-in (imported) in the notebook.
- You can use the function 'np.tanh()'. It is part of the numpy library.
- The steps you have to implement are:
  1. Retrieve each parameter from the dictionary "parameters" (which is the output of 'initialize_parameters()') by using 'parameters[".."]'.
  2. Implement Forward Propagation. Compute $Z^{[1]}, A^{[1]}, Z^{[2]}$ and $A^{[2]}$ (the vector of all your predictions on all the examples in the training set).
- Values needed in the backpropagation are stored in "cache". The *cache* will be given as an input to the backpropagation function.

```
# GRADED FUNCTION: forward_propagation

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

```
    Returns:
    A2 -- The sigmoid output of the second activation
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"
    """
    # Retrieve each parameter from the dictionary "parameters"
    ### START CODE HERE ### ( 4 lines of code)
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    ### END CODE HERE ###

    # Implement Forward Propagation to calculate A2 (probabilities)
    ### START CODE HERE ### ( 4 lines of code)
    Z1 = np.dot(W1,X)+b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2,A1)+b2
    A2 = sigmoid(Z2)
    ### END CODE HERE ###

    assert(A2.shape == (1, X.shape[1]))

    cache = {"Z1": Z1,
             "A1": A1,
             "Z2": Z2,
             "A2": A2}

    return A2, cache
```

Now that you have computed $A^{[2]}$ (in the Python variable "A2"), which contains $a^{[2](i)}$ for every example, you can compute the cost function as follows:

$$J = -\frac{1}{m}\sum_{i=0}^{m}(y^{(i)}\log\left(a^{[2](i)}\right) + (1-y^{(i)})\log\left(1-a^{[2](i)}\right)) \tag{1.3.7}$$

Exercise: Implement "compute_cost()" to compute the value of the cost J.

Instructions:

There are many ways to implement the cross-entropy loss. To help you, we give you how we would have implemented $-\sum_{i=0}^{m} y^{(i)}\log(a^{[2](i)})$:

```
logprobs = np.multiply(np.log(A2),Y)
cost = - np.sum(logprobs)                # no need to use a for loop!
```

(you can use either "np.multiply()" and then "np.sum()" or directly 'np.dot()').

```
# GRADED FUNCTION: compute_cost
def compute_cost(A2, Y, parameters):
    """
    Computes the cross-entropy cost given in equation (13)
```

```
    Arguments:
    A2 -- The sigmoid output of the second activation, of shape (1,
↪   number of examples)
    Y -- "true" labels vector of shape (1, number of examples)
    parameters -- python dictionary containing your parameters W1, b1,
↪   W2 and b2

    Returns:
    cost -- cross-entropy cost given equation (13)
    """

    m = Y.shape[1] # number of example

    # Compute the cross-entropy cost
    ### START CODE HERE ### ( 2 lines of code)
    logprobs = np.multiply(np.log(A2),Y)+np.multiply(np.log(1-A2),1-Y)
    cost = - np.sum(logprobs)/m
    ### END CODE HERE ###
#    use directly np.dot())
#    cost=-(np.dot(Y,np.log(A2.T))+np.dot(np.log(1-A2),(1-Y).T))/m

    cost = np.squeeze(cost)      # makes sure cost is the dimension we
    ↪  expect.
                                 # E.g., turns [[17]] into 17

    return cost
```

Using the cache computed during forward propagation, you can now implement **backward propagatio**.

Question: Implement the function backward_propagation().

Instructions: Backpropagation is usually the hardest (most mathematical) part in deep learning. To help you, here again is the slide from the lecture on backpropagation. You'll want to use the six equations on the right of this slide, since you are building a vectorized implementation.

$$\frac{\partial \mathcal{J}}{\partial z_2^{(i)}} = \frac{1}{m}(a^{[2](i)} - y^{(i)}) \tag{1.3.8}$$

$$\frac{\partial \mathcal{J}}{\partial W_2} = \frac{\partial \mathcal{J}}{\partial z_2^{(i)}} a^{[1](i)T} \tag{1.3.9}$$

$$\frac{\partial \mathcal{J}}{\partial b_2} = \sum_i \frac{\partial \mathcal{J}}{\partial z_2^{(i)}} \tag{1.3.10}$$

$$\frac{\partial \mathcal{J}}{\partial z_1^{(i)}} = W_2^T \frac{\partial \mathcal{J}}{\partial z_2^{(i)}} * (1 - a^{[1](i)2}) \tag{1.3.11}$$

$$\frac{\partial \mathcal{J}}{\partial W_1} = \frac{\partial \mathcal{J}}{\partial z_1^{(i)}} X^T \tag{1.3.12}$$

$$\frac{\partial \mathcal{J}_i}{\partial b_1} = \sum_i \frac{\partial \mathcal{J}}{\partial z_1^{(i)}} \tag{1.3.13}$$

- Note that $*$ denotes elementwise multiplication.
- The notation you will use is common in deep learning coding:
  - dW1 = $\frac{\partial \mathcal{J}}{\partial W_1}$
  - db1 = $\frac{\partial \mathcal{J}}{\partial b_1}$
  - dW2 = $\frac{\partial \mathcal{J}}{\partial W_2}$
  - db2 = $\frac{\partial \mathcal{J}}{\partial b_2}$

Summary of gradient descent

$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]} a^{[1]T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$	$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$
$dW^{[1]} = dz^{[1]} x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

Andrew Ng

Figure 1.3.5 Back propagation

Tips:

To compute dZ1 you'll need to compute $g^{[1]'}(Z^{[1]})$. Since $g^{[1]}(.)$ is the tanh activation function, if $a = g^{[1]}(z)$ then $g^{[1]'}(z) = 1 - a^2$. So you can compute $g^{[1]'}(Z^{[1]})$ using '(1 - np.power(A1, 2))'.

```
# GRADED FUNCTION: backward_propagation
def backward_propagation(parameters, cache, X, Y):
    """
    Implement the backward propagation using the instructions above.

    Arguments:
    parameters -- python dictionary containing our parameters
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
```

```
    X -- input data of shape (2, number of examples)
    Y -- "true" labels vector of shape (1, number of examples)

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

    # First, retrieve W1 and W2 from the dictionary "parameters".
    ### START CODE HERE ### ( 2 lines of code)
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    ### END CODE HERE ###

    # Retrieve also A1 and A2 from dictionary "cache".
    ### START CODE HERE ### ( 2 lines of code)
    A1 = cache["A1"]
    A2 = cache["A2"]
    ### END CODE HERE ###

    # Backward propagation: calculate dW1, db1, dW2, db2.
    ### START CODE HERE ### ( 6 lines of code, corresponding to 6
    ↪  equations on slide above)
    dZ2 = A2-Y
    dW2 = np.dot(dZ2,A1.T)/m
    db2 = np.sum(dZ2,axis=1,keepdims=True)/m
    dZ1 = np.dot(W2.T,dZ2)*(1 - np.power(A1, 2))
    dW1 = np.dot(dZ1,X.T)/m
    db1 = np.sum(dZ1,axis=1,keepdims=True)/m
    ### END CODE HERE ###

    grads = {"dW1": dW1,
             "db1": db1,
             "dW2": dW2,
             "db2": db2}

    return grads
```

Question: Implement the update rule. Use gradient descent. You have to use (dW1, db1, dW2, db2) in order to update (W1, b1, W2, b2).

General gradient descent rule: $\theta = \theta - \alpha \frac{\partial J}{\partial \theta}$ where α is the learning rate and θ represents a parameter.

```
# GRADED FUNCTION: update_parameters
def update_parameters(parameters, grads, learning_rate = 1.2):
    """
    Updates parameters using the gradient descent update rule given
↪   above
```

```
    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients

    Returns:
    parameters -- python dictionary containing your updated parameters
    """
    # Retrieve each parameter from the dictionary "parameters"
    ### START CODE HERE ### ( 4 lines of code)
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    ### END CODE HERE ###

    # Retrieve each gradient from the dictionary "grads"
    ### START CODE HERE ### ( 4 lines of code)
    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]
    ## END CODE HERE ###

    # Update rule for each parameter
    ### START CODE HERE ### ( 4 lines of code)
    W1 = W1-learning_rate*dW1
    b1 = b1-learning_rate*db1
    W2 = W2-learning_rate*dW2
    b2 = b2-learning_rate*db2
    ### END CODE HERE ###

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

1.3.4.4 Integrate parts 1.3.4.1, 1.3.4.2 and 1.3.4.3 in nn_model()

Question: Build your neural network model in nn_model().

Instructions: The neural network model has to use the previous functions in the right order.

```
# GRADED FUNCTION: nn_model
def nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False):
    """
```

```
    Arguments:
    X -- dataset of shape (2, number of examples)
    Y -- labels of shape (1, number of examples)
    n_h -- size of the hidden layer
    num_iterations -- Number of iterations in gradient descent loop
    print_cost -- if True, print the cost every 1000 iterations

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

    np.random.seed(3)
    n_x = layer_sizes(X, Y)[0]
    n_y = layer_sizes(X, Y)[2]

    # Initialize parameters, then retrieve W1, b1, W2, b2. Inputs:
    ↪  "n_x, n_h, n_y". Outputs = "W1, b1, W2, b2, parameters".
    ### START CODE HERE ### (  5 lines of code)
    parameters = initialize_parameters(n_x, n_h, n_y)
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    ### END CODE HERE ###

    # Loop (gradient descent)

    for i in range(0, num_iterations):

        ### START CODE HERE ### (  4 lines of code)
        # Forward propagation. Inputs: "X, parameters". Outputs: "A2,
        ↪  cache".
        A2, cache =  forward_propagation(X, parameters)

        # Cost function. Inputs: "A2, Y, parameters". Outputs: "cost".
        cost = compute_cost(A2, Y, parameters)

        # Backpropagation. Inputs: "parameters, cache, X, Y". Outputs:
        ↪  "grads".
        grads = backward_propagation(parameters, cache, X, Y)

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

        ### END CODE HERE ###
```

```
        # Print the cost every 1000 iterations
        if print_cost and i % 1000 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    return parameters
```

Question: Use your model to predict by building predict().Use forward propagation to predict results.

Reminder: predictions $= y_{prediction} = \mathbb{1}\{\text{activation} > 0.5\} = \begin{cases} 1 & \text{if } activation > 0.5 \\ 0 & \text{otherwise} \end{cases}$

As an example, if you would like to set the entries of a matrix X to 0 and 1 based on a threshold you would do: $X_{new} = (X > threshold)$.

```
# GRADED FUNCTION: predict
def predict(parameters, X):
    """
    Using the learned parameters, predicts a class for each example in
 ↪  X

    Arguments:
    parameters -- python dictionary containing your parameters
    X -- input data of size (n_x, m)

    Returns
    predictions -- vector of predictions of our model (red: 0 / blue:
 ↪  1)
    """

    # Computes probabilities using forward propagation, and classifies
     ↪  to 0/1 using 0.5 as the threshold.
    ### START CODE HERE ### ( 2 lines of code)
    A2, cache = forward_propagation(X, parameters)
    predictions = np.round(A2)
    ### END CODE HERE ###

    return predictions
```

It is time to run the model and see how it performs on a planar dataset. Run the following code to test your model with a single hidden layer of n_h hidden units.

```
# Build a model with a n_h-dimensional hidden layer
parameters = nn_model(X, Y, n_h = 4, num_iterations = 10000,
 ↪  print_cost=True)

# Plot the decision boundary
plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
plt.title("Decision Boundary for hidden layer size " + str(4))
```

The classification result of planar dataset is as follows:

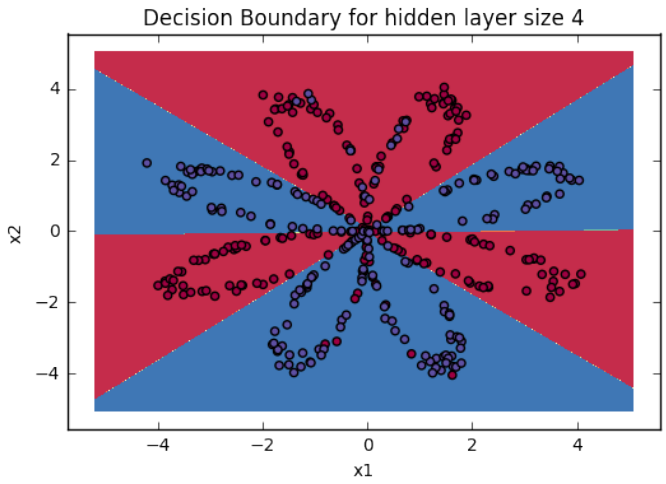


Figure 1.3.6 Decision Boundary for hidden layer size

```
# Print accuracy
predictions = predict(parameters, X)
print ('Accuracy: %d' % float((np.dot(Y,predictions.T) +
↪  np.dot(1-Y,1-predictions.T))/float(Y.size)*100) + '%')

#Accuracy
Accuracy: 90%
```

Accuracy is really high compared to Logistic Regression. The model has learnt the leaf patterns of the flower! Neural networks are able to learn even highly non-linear decision boundaries, unlike logistic regression.

Now, let's try out several hidden layer sizes.

1.3.4.5 Tuning hidden layer size (optional/ungraded exercise)

Run the following code. It may take 1-2 minutes. You will observe different behaviors of the model for various hidden layer sizes.

```
# This may take about 2 minutes to run

plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 20, 50]
for i, n_h in enumerate(hidden_layer_sizes):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer of size %d' % n_h)
    parameters = nn_model(X, Y, n_h, num_iterations = 5000)
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    predictions = predict(parameters, X)
    accuracy = float((np.dot(Y,predictions.T) +
    ↪  np.dot(1-Y,1-predictions.T))/float(Y.size)*100)
    print ("Accuracy for {} hidden units: {} %".format(n_h, accuracy))
```

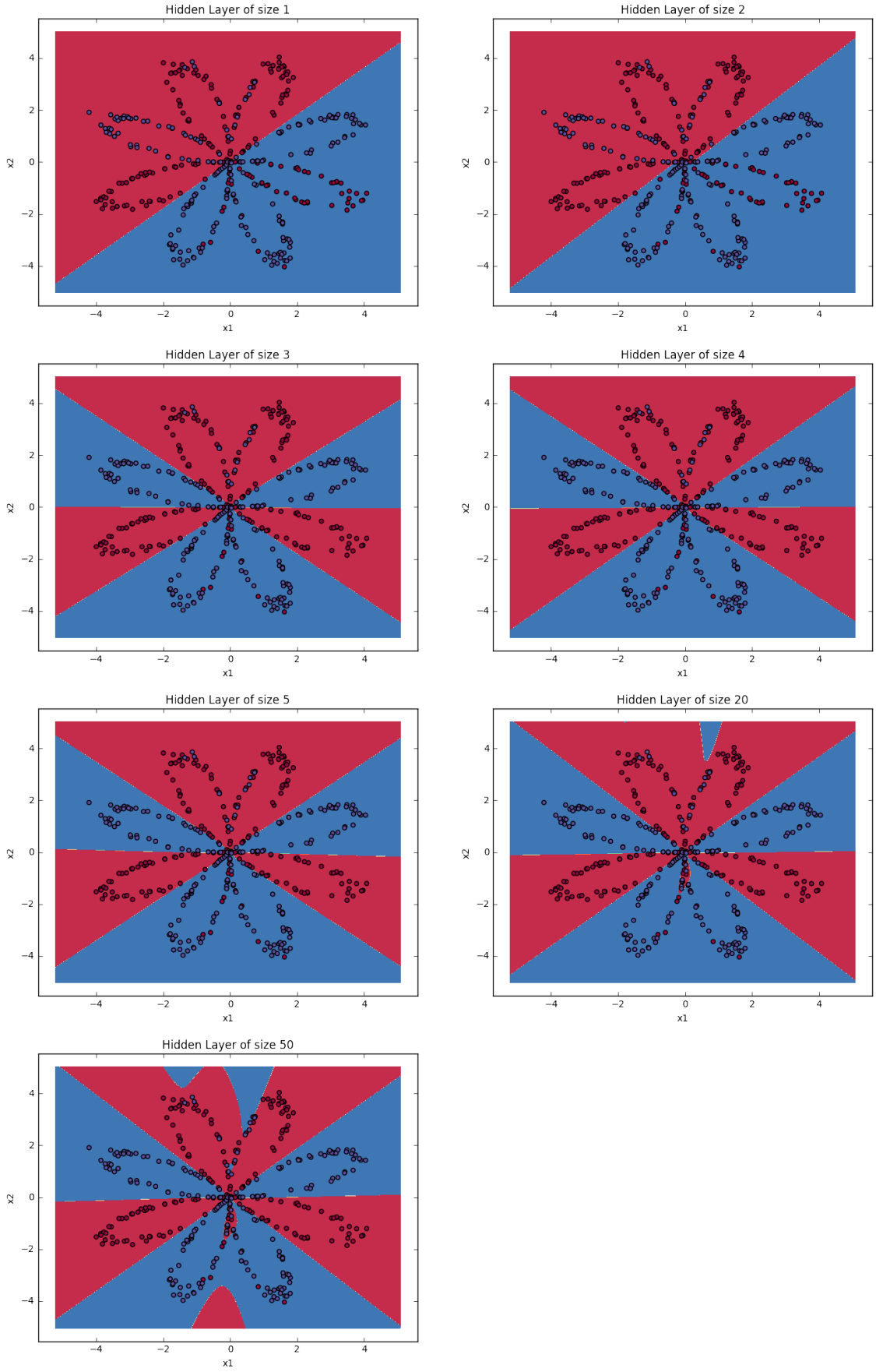


Figure 1.3.7 Different behaviors of the model for various hidden layer sizes

Interpretation:

- The larger models (with more hidden units) are able to fit the training set better, until eventually the largest models overfit the data.
- The best hidden layer size seems to be around $n_h = 5$. Indeed, a value around here seems to fits the data well without also incurring noticable overfitting.
- You will also learn later about regularization, which lets you use very large models (such as $n_h = 50$) without much overfitting.

You've learnt to:

- **Build a complete neural network with a hidden layer**
- **Make a good use of a non-linear unit**
- **Implemented forward propagation and backpropagation, and trained a neural network**
- **See the impact of varying the hidden layer size, including overfitting.**

1.3.5 Code of Neural Network With a Hidden Layer

```
# Package imports
import numpy as np
import matplotlib.pyplot as plt
from testCases_v2 import *
import sklearn
import sklearn.datasets
import sklearn.linear_model
from planar_utils import plot_decision_boundary, sigmoid,
↪  load_planar_dataset, load_extra_datasets

#matplotlib inline

np.random.seed(1) # set a seed so that the results are consistent

X, Y = load_planar_dataset()

# Visualize the data:
#plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);

shape_X = X.shape
shape_Y = Y.shape
m = shape_X[1]  # training set size

# Train the logistic regression classifier
#===================================
# clf = sklearn.linear_model.LogisticRegressionCV();
# clf.fit(X.T, Y.T);
#
# # Plot the decision boundary for logistic regression
# plot_decision_boundary(lambda x: clf.predict(x), X, Y)
# plt.title("Logistic Regression")
# # Print accuracy
# LR_predictions = clf.predict(X.T)
# print ('Accuracy of logistic regression: %d '
↪  %float((np.dot(Y,LR_predictions)
↪  +np.dot(1-Y,1-LR_predictions))/float(Y.size)*100) +'% ' +
↪  "(percentage of correctly labelled datapoints)")
#===================================

# GRADED FUNCTION: layer_sizes
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_h -- the size of the hidden layer
    n_y -- the size of the output layer
    """
    n_x = X.shape[0] # size of input layer
    n_h = 4
    n_y = Y.shape[0]# size of output layer
    return (n_x, n_h, n_y)


# GRADED FUNCTION: initialize_parameters
def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer

    Returns:
    params -- python dictionary containing your parameters:
                    W1 -- weight matrix of shape (n_h, n_x)
                    b1 -- bias vector of shape (n_h, 1)
                    W2 -- weight matrix of shape (n_y, n_h)
                    b2 -- bias vector of shape (n_y, 1)
    """

    np.random.seed(2) # we set up a seed so that your output matches
     ↪  ours although the initialization is random.

    W1 = np.random.randn(n_h,n_x) * 0.01
    b1 = np.zeros((n_h,1))
    W2 = np.random.randn(n_y,n_h)* 0.01
    b2 = np.zeros((n_y,1))

    assert (W1.shape == (n_h, n_x))
    assert (b1.shape == (n_h, 1))
    assert (W2.shape == (n_y, n_h))
    assert (b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters


# GRADED FUNCTION: forward_propagation
```

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

    Returns:
    A2 -- The sigmoid output of the second activation
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"
    """
    # Retrieve each parameter from the dictionary "parameters"
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Implement Forward Propagation to calculate A2 (probabilities)
    Z1 = np.dot(W1,X)+b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2,A1)+b2
    A2 = sigmoid(Z2)

    assert(A2.shape == (1, X.shape[1]))

    cache = {"Z1": Z1,
             "A1": A1,
             "Z2": Z2,
             "A2": A2}

    return A2, cache


# GRADED FUNCTION: compute_cost
def compute_cost(A2, Y, parameters):
    """
    Computes the cross-entropy cost given in equation (13)

    Arguments:
    A2 -- The sigmoid output of the second activation, of shape (1,
↪   number of examples)
    Y -- "true" labels vector of shape (1, number of examples)
    parameters -- python dictionary containing your parameters W1, b1,
↪   W2 and b2

    Returns:
    cost -- cross-entropy cost given equation (13)
    """
```

```
    m = Y.shape[1] # number of example

    # Compute the cross-entropy cost
    logprobs = np.multiply(np.log(A2),Y)+np.multiply(np.log(1-A2),1-Y)
    cost = - np.sum(logprobs)/m

#    use directly np.dot())
#    cost=-(np.dot(Y,np.log(A2.T))+np.dot(np.log(1-A2),(1-Y).T))/m

    cost = np.squeeze(cost)      # makes sure cost is the dimension we
    ↪  expect.
                                 # E.g., turns [[17]] into 17

    return cost


# GRADED FUNCTION: backward_propagation
def backward_propagation(parameters, cache, X, Y):
    """
    Implement the backward propagation using the instructions above.

    Arguments:
    parameters -- python dictionary containing our parameters
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
    X -- input data of shape (2, number of examples)
    Y -- "true" labels vector of shape (1, number of examples)

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

    # First, retrieve W1 and W2 from the dictionary "parameters".
    W1 = parameters["W1"]
    W2 = parameters["W2"]

    # Retrieve also A1 and A2 from dictionary "cache".
    A1 = cache["A1"]
    A2 = cache["A2"]

    # Backward propagation: calculate dW1, db1, dW2, db2.
    dZ2 = A2-Y
    dW2 = np.dot(dZ2,A1.T)/m
    db2 = np.sum(dZ2,axis=1,keepdims=True)/m
    dZ1 = np.dot(W2.T,dZ2)*(1 - np.power(A1, 2))
    dW1 = np.dot(dZ1,X.T)/m
```

```
    db1 = np.sum(dZ1,axis=1,keepdims=True)/m

    grads = {"dW1": dW1,
             "db1": db1,
             "dW2": dW2,
             "db2": db2}

    return grads


# GRADED FUNCTION: update_parameters
def update_parameters(parameters, grads, learning_rate = 1.2):
    """
    Updates parameters using the gradient descent update rule given
 ↪  above

    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients

    Returns:
    parameters -- python dictionary containing your updated parameters
    """
    # Retrieve each parameter from the dictionary "parameters"
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Retrieve each gradient from the dictionary "grads"
    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]

    # Update rule for each parameter
    W1 = W1-learning_rate*dW1
    b1 = b1-learning_rate*db1
    W2 = W2-learning_rate*dW2
    b2 = b2-learning_rate*db2

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

```
# GRADED FUNCTION: nn_model
def nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False):
    """
    Arguments:
    X -- dataset of shape (2, number of examples)
    Y -- labels of shape (1, number of examples)
    n_h -- size of the hidden layer
    num_iterations -- Number of iterations in gradient descent loop
    print_cost -- if True, print the cost every 1000 iterations

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

    np.random.seed(3)
    n_x = layer_sizes(X, Y)[0]
    n_y = layer_sizes(X, Y)[2]

    # Initialize parameters, then retrieve W1, b1, W2, b2. Inputs:
    ↪  "n_x, n_h, n_y". Outputs = "W1, b1, W2, b2, parameters".
    parameters = initialize_parameters(n_x, n_h, n_y)
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Loop (gradient descent)

    for i in range(0, num_iterations):

        # Forward propagation. Inputs: "X, parameters". Outputs: "A2,
        ↪  cache".
        A2, cache =  forward_propagation(X, parameters)

        # Cost function. Inputs: "A2, Y, parameters". Outputs: "cost".
        cost = compute_cost(A2, Y, parameters)

        # Backpropagation. Inputs: "parameters, cache, X, Y". Outputs:
        ↪  "grads".
        grads = backward_propagation(parameters, cache, X, Y)

        # Gradient descent parameter update. Inputs: "parameters,
        ↪  grads". Outputs: "parameters".
        parameters = update_parameters(parameters, grads)
```

```
        # Print the cost every 1000 iterations
        if print_cost and i % 1000 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    return parameters


#GRADED FUNCTION: predict
def predict(parameters, X):
    """
    Using the learned parameters, predicts a class for each example in
 ↪  X

    Arguments:
    parameters -- python dictionary containing your parameters
    X -- input data of size (n_x, m)

    Returns
    predictions -- vector of predictions of our model (red: 0 / blue:
 ↪  1)
    """

    # Computes probabilities using forward propagation, and classifies
     ↪  to 0/1 using 0.5 as the threshold.
    A2, cache = forward_propagation(X, parameters)
    predictions = np.round(A2)

    return predictions


# Build a model with a n_h-dimensional hidden layer
parameters = nn_model(X, Y, n_h = 4, num_iterations = 10000,
 ↪  print_cost=True)
plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
plt.title("Decision Boundary for hidden layer size " + str(4))
print ('Accuracy: %d' % float((np.dot(Y,predictions.T) +
 ↪  np.dot(1-Y,1-predictions.T))/float(Y.size)*100) + '%')
```

1.4 Building your Deep Neural Network: Step by Step

Welcome to your third programming exercise of the deep learning specialization. You will implement all the building blocks of a neural network and use these building blocks in the next assignment to build a neural network of any architecture you want. By completing this assignment you will:

- Develop an intuition of the over all structure of a neural network.
- Write functions (e.g. forward propagation, backward propagation, logistic loss, etc...) that would help you decompose your code and ease the process of building a neural network.
- Initialize/update parameters according to your desired structure.

This assignment prepares you well for the upcoming assignment. In the next assignment, you will use these functions to build a deep neural network for image classification. Take your time to complete it and make sure you get the expected outputs when working through the different exercises. In some code blocks, you will find a "#GRADED FUNCTION: functionName" comment. Please do not modify it. After you are done, submit your work and check your results. You need to score 70% to pass. Good luck :) !

After this assignment you will be able to:

- Use non-linear units like ReLU to improve your model
- Build a deeper neural network (with more than 1 hidden layer)
- Implement an easy-to-use neural network class

Notation:

- Superscript $[l]$ denotes a quantity associated with the l^{th} layer.
  - Example: $a^{[L]}$ is the L^{th} layer activation. $W^{[L]}$ and $b^{[L]}$ are the L^{th} layer parameters.
- Superscript (i) denotes a quantity associated with the i^{th} example.
  - Example: $x^{(i)}$ is the i^{th} training example.
- Lowerscript i denotes the i^{th} entry of a vector.
  - Example: $a_i^{[l]}$ denotes the i^{th} entry of the l^{th} layer's activations).

Let's get started!

1.4.1 Packages

Let's first import all the packages that you will need during this assignment.

- numpy is the main package for scientific computing with Python.
- matplotlib is a library to plot graphs in Python.
- dnn_utils provides some necessary functions for this notebook.
- testCases provides some test cases to assess the correctness of your functions
- np.random.seed(1) is used to keep all the random function calls consistent. It will help us grade your work. Please don't change the seed.

1.4.2 Outline of the Assignment

To build your neural network, you will be implementing several"helper functions". These helper functions will be used in the next assignment to build a two-layer neural network and an L-layer neural network. Each small helper function you will implement will have detailed instructions that will walk you through the necessary steps. Figure 1.4.2 is an outline of this assignment, you will:

- Initialize the parameters for a two-layer network and for an L-layer neural network.
- Implement the forward propagation module (shown in purple in the figure below).
  - Complete the LINEAR part of a layer's forward propagation step (resulting in $Z^{[l]}$).
  - We give you the ACTIVATION function (relu/sigmoid).
  - Combine the previous two steps into a new [LINEAR->ACTIVATION] forward function.
  - Stack the [LINEAR->RELU] forward function L-1 time (for layers 1 through L-1) and add a [LINEAR->SIGMOID] at the end (for the final layer L). This gives you a new L_model_forward function.
- Compute the loss.
- Implement the backward propagation module (denoted in red in the figure below).
  - Complete the LINEAR part of a layer's backward propagation step.
  - We give you the gradient of the ACTIVATE function (relu_backward/sigmoid_backward)
  - Combine the previous two steps into a new [LINEAR->ACTIVATION] backward function.
  - Stack [LINEAR->RELU] backward L-1 times and add [LINEAR->SIGMOID] backward in a new L_model_backward function
- Finally update the parameters.

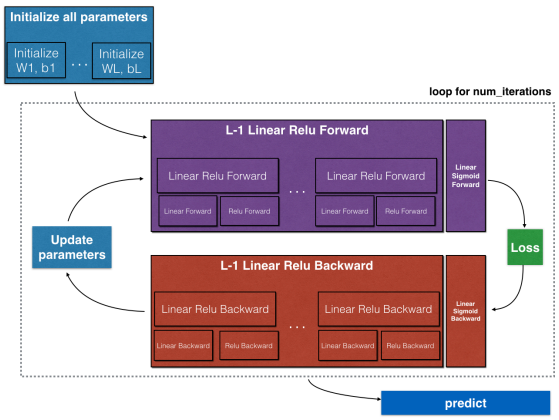


Figure 1.4.1 Outline

Note that for every forward function, there is a corresponding backward function. That is why at every step of your forward module you will be storing some values in a cache. The cached values are useful for computing gradients. In the backpropagation module you will then use the cache to calculate the gradients. This assignment will show you exactly how to carry out each of these steps.

1.4.3 Initialization

You will write two helper functions that will initialize the parameters for your model. The first function will be used to initialize parameters for a two layer model. The second one will generalize this initialization process to L layers.

1.4.3.1 2-layer Neural Network

Exercise: Create and initialize the parameters of the 2-layer neural network.
Instructions:

- The model's structure is: LINEAR -> RELU -> LINEAR -> SIGMOID.
- Use random initialization for the weight matrices. Use np.random.randn(shape)*0.01 with the correct shape.
- Use zero initialization for the biases. Use np.zeros(shape).

```
# GRADED FUNCTION: initialize_parameters
def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer

    Returns:
    parameters -- python dictionary containing your parameters:
                    W1 -- weight matrix of shape (n_h, n_x)
                    b1 -- bias vector of shape (n_h, 1)
                    W2 -- weight matrix of shape (n_y, n_h)
                    b2 -- bias vector of shape (n_y, 1)
    """

    np.random.seed(1)

    ### START CODE HERE ### ( 4 lines of code)
    W1 = np.random.randn(n_h, n_x)*0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h)*0.01
    b2 = np.zeros((n_y, 1))
    ### END CODE HERE ###

    assert(W1.shape == (n_h, n_x))
```

```
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

1.4.3.2 L-layer Neural Network

The initialization for a deeper L-layer neural network is more complicated because there are many more weight matrices and bias vectors. When completing the "initialize_parameters_deep", you should make sure that your dimensions match between each layer. Recall that $n^{[l]}$ is the number of units in layer l. Thus for example if the size of our input X is $(12288, 209)$ (with $m = 209$ examples) then:

	Shape of W	Shape of b	Activation	Shape of Activation
Layer 1	$(n^{[1]}, 12288)$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]}X + b^{[1]}$	$(n^{[1]}, 209)$
Layer 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$	$(n^{[2]}, 209)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Layer L-1	$(n^{[L-1]}, n^{[L-2]})$	$(n^{[L-1]}, 1)$	$Z^{[L-1]} = W^{[L-1]}A^{[L-2]} + b^{[L-1]}$	$(n^{[L-1]}, 209)$
Layer L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 209)$

Remember that when we compute $WX + b$ in python, it carries out broadcasting. For example, if:

$$W = \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} \quad X = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad b = \begin{bmatrix} s \\ t \\ u \end{bmatrix}$$

Then $WX + b$ will be:

$$WX + b = \begin{bmatrix} (ja + kd + lg) + s & (jb + ke + lh) + s & (jc + kf + li) + s \\ (ma + nd + og) + t & (mb + ne + oh) + t & (mc + nf + oi) + t \\ (pa + qd + rg) + u & (pb + qe + rh) + u & (pc + qf + ri) + u \end{bmatrix} \tag{1.4.1}$$

Exercise: Implement initialization for an L-layer Neural Network.
Instructions:

- The model's structure is [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID. I.e., it has $L-1$ layers using a ReLU activation function followed by an output layer with a sigmoid activation function.
- Use random initialization for the weight matrices. Use "np.random.rand(shape) * 0.01".
- Use zeros initialization for the biases. Use "np.zeros(shape)".

- We will store $n^{[l]}$, the number of units in different layers, in a variable "layer_dims". For example, the "layer_dims" for the "Planar Data classification mode" from last week would have been [2,4,1]: There were two inputs, one hidden layer with 4 hidden units, and an output layer with 1 output unit. Thus means "W1"'s shape was (4,2), "b1" was (4,1), "W2" was (1,4) and "b2" was (1,1). Now you will generalize this to L layers!
- Here is the implementation for $L = 1$ (one layer neural network). It should inspire you to implement the general case (L-layer neural network).

```
    if L == 1:
        parameters["W" + str(L)] = np.random.randn(layer_dims[1],
        ↪  layer_dims[0]) * 0.01
        parameters["b" + str(L)] = np.zeros((layer_dims[1], 1))
```

```
# GRADED FUNCTION: initialize_parameters_deep
def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each
↪   layer in our network

    Returns:
    parameters -- python dictionary containing your parameters "W1",
↪   "b1", ..., "WL", "bL":
      Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
      bl -- bias vector of shape (layer_dims[l], 1)
    """

    np.random.seed(3)
    parameters = {}
    L = len(layer_dims)            # number of layers in the network

    for l in range(1, L):
        ### START CODE HERE ### ( 2 lines of code)
        parameters['W' + str(l)] = np.random.randn(layer_dims[l],
        ↪  layer_dims[l-1]) * 0.01
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))
        ### END CODE HERE ###

        assert(parameters['W' + str(l)].shape == (layer_dims[l],
        ↪  layer_dims[l-1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))


    return parameters
```

1.4.4 Forward propagation module

1.4.4.1 Linear Forward

Now that you have initialized your parameters, you will do the forward propagation module. You will start by implementing some basic functions that you will use later when implementing the model. You will complete three functions in this order:

1 LINEAR

2 LINEAR -> ACTIVATION where ACTIVATION will be either ReLU or Sigmoid

3 [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID (whole model)

The linear forward module (vectorized over all the examples) computes the following equations:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \tag{1.4.2}$$

where $A^{[0]} = X$.

Exercise: Build the linear part of forward propagation.

Reminder: The mathematical representation of this unit is $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$. You may also find "np.dot()" useful. If your dimensions don't match, printing "W.shape" may help.

```
# GRADED FUNCTION: linear_forward
def linear_forward(A, W, b):
    """
    Implement the linear part of a layer's forward propagation.

    Arguments:
    A -- activations from previous layer (or input data): (size of
↪   previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer,
↪   size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer,
↪   1)

    Returns:
    Z -- the input of the activation function, also called
↪   pre-activation parameter
    cache -- a python dictionary containing "A", "W" and "b" ; stored
↪   for computing the backward pass efficiently
    """

    ### START CODE HERE ### ( 1 line of code)
    Z = np.dot(W,A)+b
    ### END CODE HERE ###

    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache
```

1.4.4.2 Linear-Activation Forward

In this notebook, you will use two activation functions:

Sigmoid: $\sigma(Z) = \sigma(WA+b) = \frac{1}{1+e^{-(WA+b)}}$. We have provided you with the "sigmoid" function. This function returns **two** items: the activation value "a" and a "cache" that contains "Z" (it's what we will feed in to the corresponding backward function). To use it you could just call:

```
A, activation_cache = sigmoid(Z)
```

ReLU: The mathematical formula for ReLu is $A = RELU(Z) = max(0, Z)$. We have provided you with the "relu" function. This function returns **two** items: the activation value "A" and a "cache" that contains "Z" (it's what we will feed in to the corresponding backward function). To use it you could just call:

```
A, activation_cache = relu(Z)
```

For more convenience, you are going to group two functions (Linear and Activation) into one function (LINEAR->ACTIVATION). Hence, you will implement a function that does the LINEAR forward step followed by an ACTIVATION forward step.

Exercise: Implement the forward propagation of the *LINEAR->ACTIVATION* layer. Mathematical relation is: $A^{[l]} = g(Z^{[l]}) = g(W^{[l]}A^{[l-1]} + b^{[l]})$ where the activation "g" can be sigmoid() or relu(). Use linear_forward() and the correct activation function.

```
# GRADED FUNCTION: linear_activation_forward
def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR->ACTIVATION layer

    Arguments:
    A_prev -- activations from previous layer (or input data): (size of
↪   previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer,
↪   size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer,
↪   1)
    activation -- the activation to be used in this layer, stored as a
↪   text string: "sigmoid" or "relu"

    Returns:
    A -- the output of the activation function, also called the
↪   post-activation value
    cache -- a python dictionary containing "linear_cache" and
↪   "activation_cache";
             stored for computing the backward pass efficiently
    """
```

```
    if activation == "sigmoid":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        ### START CODE HERE ### ( 2 lines of code)
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)
        ### END CODE HERE ###

    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        ### START CODE HERE ### ( 2 lines of code)
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)
        ### END CODE HERE ###

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache
```

Note: In deep learning, the *[LINEAR->ACTIVATION]* computation is counted as a single layer in the neural network, not two layers.

1.4.4.3 L-Layer Model

For even more convenience when implementing the L-layer Neural Net, you will need a function that replicates the previous one (linear_activation_forward with RELU) $L-1$ times, then follows that with one linear_activation_forward with SIGMOID.

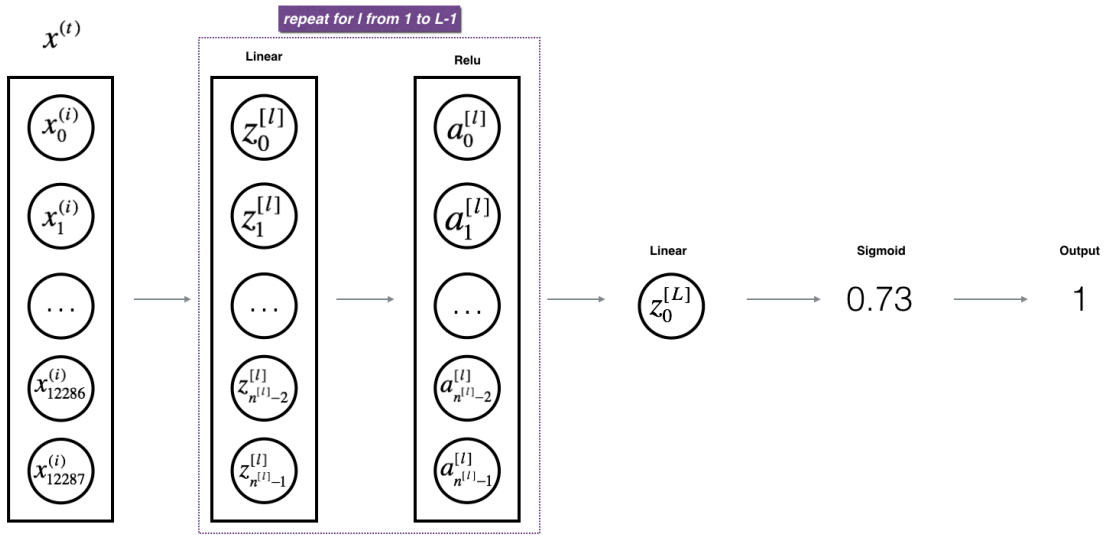


Figure 1.4.2 [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID model

Exercise: Implement the forward propagation of the above model.

Instruction: In the code below, the variable "AL" will denote $A^{[L]} = \sigma(Z^{[L]}) = \sigma(W^{[L]}A^{[L-1]} + b^{[L]})$. (This is sometimes also called "Yhat", i.e., this is $\hat{Y}$.)

Tips:

- Use the functions you had previously written
- Use a for loop to replicate [LINEAR->RELU] (L-1) times
- Don't forget to keep track of the caches in the "caches" list. To add a new value "c" to a "list", you can use "list.append(c)".

```
# GRADED FUNCTION: L_model_forward
def L_model_forward(X, parameters):
    """
    Implement forward propagation for the
↪   [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID computation

    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    parameters -- output of initialize_parameters_deep()

    Returns:
    AL -- last post-activation value
    caches -- list of caches containing:
                every cache of linear_relu_forward() (there are L-1 of
↪   them, indexed from 0 to L-2)
                the cache of linear_sigmoid_forward() (there is one,
↪   indexed L-1)
    """

    caches = []
    A = X
    L = len(parameters) // 2 # number of layers in the neural network

    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches"
    ↪  list.
    for l in range(1, L):
        A_prev = A
        ### START CODE HERE ### ( 2 lines of code)
        A, cache = linear_activation_forward(A_prev, parameters['W' +
        ↪  str(l)], parameters['b' + str(l)], activation ="relu")
        caches.append(cache)
        ### END CODE HERE ###

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    ### START CODE HERE ### ( 2 lines of code)
    AL, cache = linear_activation_forward(A, parameters['W' + str(L)],
    ↪  parameters['b' + str(L)], activation = "sigmoid")
    caches.append(cache)
    ### END CODE HERE ###
```

```
    assert(AL.shape == (1,X.shape[1]))

    return AL, caches
```

Great! Now you have a full forward propagation that takes the input X and outputs a row vector $A^{[L]}$ containing your predictions. It also records all intermediate values in "caches". Using $A^{[L]}$, you can compute the cost of your predictions.

1.4.5 Cost function

Now you will implement forward and backward propagation. You need to compute the cost, because you want to check if your model is actually learning.

Exercise: Compute the cross-entropy cost J, using the following formula:

$$\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log\left(a^{[L](i)}\right)+(1-y^{(i)})\log\left(1-a^{[L](i)}\right)) \tag{1.4.3}$$

```
# GRADED FUNCTION: compute_cost
def compute_cost(AL, Y):
    """
    Implement the cost function defined by equation (7).

    Arguments:
    AL -- probability vector corresponding to your label predictions,
 ↪  shape (1, number of examples)
    Y -- true "label" vector (for example: containing 0 if non-cat, 1
 ↪  if cat), shape (1, number of examples)

    Returns:
    cost -- cross-entropy cost
    """

    m = Y.shape[1]

    # Compute loss from aL and y.
    ### START CODE HERE ### (  1 lines of code)
    cost = -(np.dot(Y,np.log(AL.T))+np.dot(1-Y,np.log(1-AL).T))/m
    ### END CODE HERE ###

    cost = np.squeeze(cost)       # To make sure your cost's shape is
    ↪  what we expect (e.g. this turns [[17]] into 17).
    assert(cost.shape == ())

    return cost
```

1.4.6 Backward propagation module

Just like with forward propagation, you will implement helper functions for backpropagation. Remember that back propagation is used to calculate the gradient of the loss function with respect to the parameters.

Reminder:

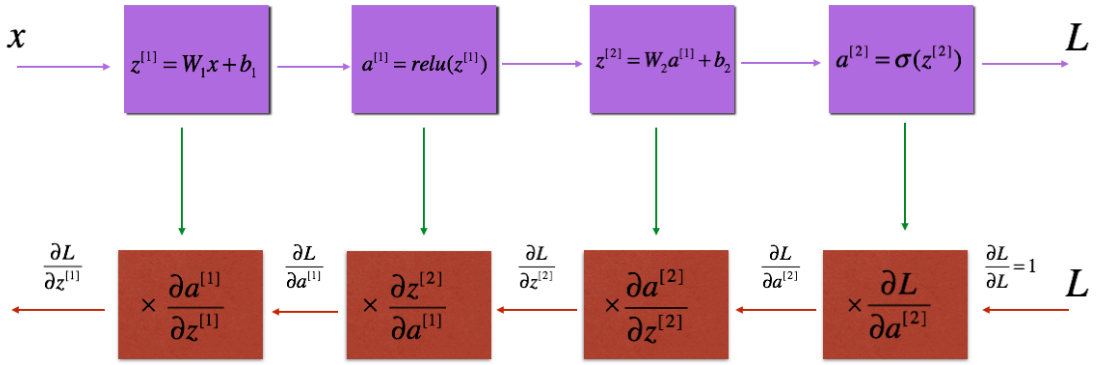


Figure 1.4.3 Forward and Backward propagation for LINEAR->RELU->LINEAR->SIGMOID. The purple blocks represent the forward propagation, and the red blocks represent the backward propagation.

Now, similar to forward propagation, you are going to build the backward propagation in three steps:

1 LINEAR backward

2 LINEAR -> ACTIVATION backward where ACTIVATION computes the derivative of either the ReLU or sigmoid activation

3 [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID backward (whole model)

1.4.6.1 Linear backward

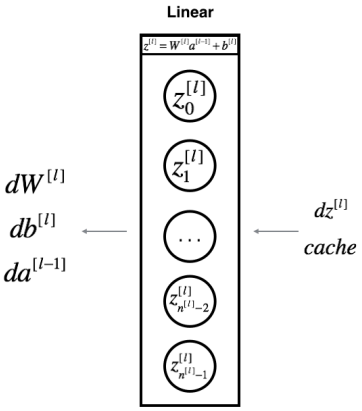


Figure 1.4.4 Linear backward

For layer l, the linear part is: $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$ (followed by an activation).

Suppose you have already calculated the derivative $dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$. You want to get $(dW^{[l]}, db^{[l]}dA^{[l-1]})$.

The three outputs $(dW^{[l]}, db^{[l]}, dA^{[l]})$ are computed using the input $dZ^{[l]}$.Here are the formulas you need:

$$dW^{[l]} = \frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \tag{1.4.4}$$

$$db^{[l]} = \frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)} \tag{1.4.5}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]} \tag{1.4.6}$$

Exercise: Use the 3 formulas above to implement linear_backward().

```
# GRADED FUNCTION: linear_backward

def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single
↪   layer (layer l)

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of
↪   current layer l)
    cache -- tuple of values (A_prev, W, b) coming from the forward
↪   propagation in the current layer

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of
↪   the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l),
↪   same shape as W
    db -- Gradient of the cost with respect to b (current layer l),
↪   same shape as b
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    ### START CODE HERE ### ( 3 lines of code)
    dW = np.dot(dZ,A_prev.T)/m
    db = np.sum(dZ,axis=1,keepdims=True)/m
    dA_prev = np.dot(W.T,dZ)
    ### END CODE HERE ###

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db
```

1.4.6.2 Linear-Activation backward

Next, you will create a function that merges the two helper functions: **linear_backward** and the backward step for the activation **linear_activation_backward**.

To help you implement *linear_activation_backward*, we provided two backward functions:

- **sigmoid_backward**: Implements the backward propagation for SIGMOID unit. You can call it as follows:

```
dZ = sigmoid_backward(dA, activation_cache)
```

- **relu_backward**: Implements the backward propagation for RELU unit. You can call it as follows:

```
dZ = relu_backward(dA, activation_cache)
```

If $g(.)$ is the activation function, *sigmoid_backward* and *relu_backward* compute

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]}) \tag{1.4.7}$$

Exercise: Implement the backpropagation for the *LINEAR->ACTIVATION* layer.

```
# GRADED FUNCTION: linear_activation_backward
def linear_activation_backward(dA, cache, activation):
    """
    Implement the backward propagation for the LINEAR->ACTIVATION
↪   layer.

    Arguments:
    dA -- post-activation gradient for current layer l
    cache -- tuple of values (linear_cache, activation_cache) we store
↪   for computing backward propagation efficiently
    activation -- the activation to be used in this layer, stored as a
↪   text string: "sigmoid" or "relu"

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of
↪   the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l),
↪   same shape as W
    db -- Gradient of the cost with respect to b (current layer l),
↪   same shape as b
    """
    linear_cache, activation_cache = cache

    if activation == "relu":
```

```
        ### START CODE HERE ### ( 2 lines of code)
        dZ =  relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
        ### END CODE HERE ###

    elif activation == "sigmoid":
        ### START CODE HERE ### ( 2 lines of code)
        dZ =  sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
        ### END CODE HERE ###

    return dA_prev, dW, db
```

1.4.6.3 L-Model Backward

Now you will implement the backward function for the whole network. Recall that when you implemented the *L_model_forward* function, at each iteration, you stored a cache which contains (X,W,b, and z). In the back propagation module, you will use those variables to compute the gradients. Therefore, in the *L_model_backward* function, you will iterate through all the hidden layers backward, starting from layer L. On each step, you will use the cached values for layer l to backpropagate through layer l. Figure 1.4.5 below shows the backward pass.

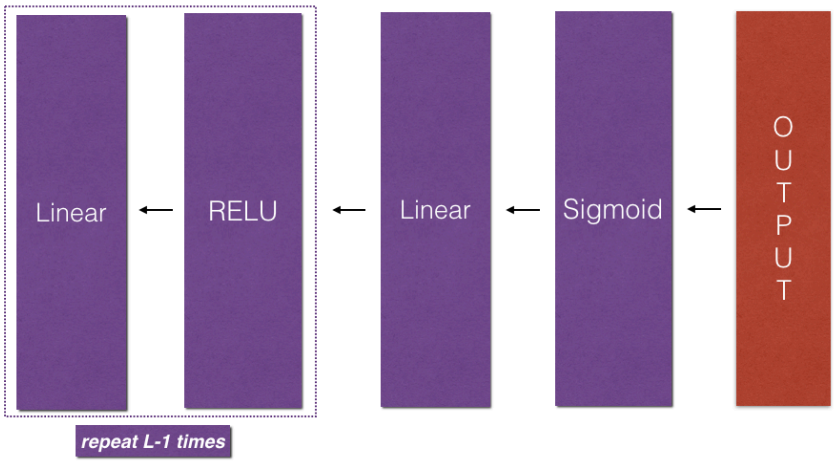


Figure 1.4.5 Backward pass

Initializing backpropagation: To backpropagate through this network, we know that the output is, $A^{[L]} = \sigma(Z^{[L]})$. Your code thus needs to compute $dAL = \frac{\partial \mathcal{L}}{\partial A^{[L]}}$. To do so, use this formula (derived using calculus which you don't need in-depth knowledge of):

```
dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL)) # derivative of
↪  cost with respect to AL
```

You can then use this post-activation gradient dAL to keep going backward. As seen in Figure 1.4.5, you can now feed in dAL into the LINEAR->SIGMOID backward function you implemented (which will use the cached values stored by the L_model_forward function). After that, you will have to use a *for* loop to iterate through all the other layers using the LINEAR->RELU backward function. You should store each dA, dW, and db in the grads dictionary. To do so, use this formula :

$$grads[\text{“}dW\text{”} + str(l)] = dW^{[l]} \tag{1.4.8}$$

For example, for $l = 3$ this would store $dW^{[l]}$ in $grads[\text{“}dW3\text{”}]$.

Exercise: Implement backpropagation for the *[LINEAR->RELU] × (L-1) -> LINEAR -> SIGMOID* model.

```
# GRADED FUNCTION: L_model_backward
def L_model_backward(AL, Y, caches):
    """
    Implement the backward propagation for the [LINEAR->RELU] * (L-1)
↪   -> LINEAR -> SIGMOID group

    Arguments:
    AL -- probability vector, output of the forward propagation
↪   (L_model_forward())
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
    caches -- list of caches containing:
                every cache of linear_activation_forward() with "relu"
↪   (it's caches[l], for l in range(L-1) i.e l = 0...L-2)
                the cache of linear_activation_forward() with "sigmoid"
↪   (it's caches[L-1])

    Returns:
    grads -- A dictionary with the gradients
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as
    ↪ AL

    # Initializing the backpropagation
    ### START CODE HERE ### (1 line of code)
    dAL =  - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
    ### END CODE HERE ###

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "AL, Y, caches".
    ↪ Outputs: "grads["dAL"], grads["dWL"], grads["dbL"]
    ### START CODE HERE ### (approx. 2 lines)
    current_cache = caches[L-1]
```

```
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] =
    ↪ linear_activation_backward(dAL, current_cache, "sigmoid")
    ### END CODE HERE ###

    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        # Inputs: "grads["dA" + str(l + 2)], caches". Outputs:
        ↪ "grads["dA" + str(l + 1)] , grads["dW" + str(l + 1)] ,
        ↪ grads["db" + str(l + 1)]
        ### START CODE HERE ### (approx. 5 lines)
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp =
        ↪ linear_activation_backward(grads["dA" + str(l+2)],
        ↪ current_cache, "relu")
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp
        ### END CODE HERE ###

    return grads
```

1.4.6.4 Update Parameters

In this section you will update the parameters of the model, using gradient descent:

$$W^{[l]} = W^{[l]} - \alpha \, dW^{[l]} \tag{1.4.9}$$

$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]} \tag{1.4.10}$$

where α is the learning rate. After computing the updated parameters, store them in the parameters dictionary.

Exercise: Implement "update_parameters()" to update your parameters using gradient descent.

Instructions: Update parameters using gradient descent on every $W^{[l]}$ and $b^{[l]}$ for $l = 1, 2, ..., L$.

```
# GRADED FUNCTION: update_parameters
def update_parameters(parameters, grads, learning_rate):
    """
    Update parameters using gradient descent

    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients, output of
↪   L_model_backward

    Returns:
    parameters -- python dictionary containing your updated parameters
                  parameters["W" + str(l)] = ...
```

```
                parameters["b" + str(l)] = ...
    """

    L = len(parameters) // 2 # number of layers in the neural network

    # Update rule for each parameter. Use a for loop.
    ### START CODE HERE ### ( 3 lines of code)
    for l in range(L):
        parameters["W" + str(l+1)] =  parameters["W" + str(l+1)] -
        ↪  learning_rate * grads["dW" + str(l + 1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
        ↪  learning_rate * grads["db" + str(l + 1)]
    ### END CODE HERE ###
    return parameters
```

1.4.6.5 Conclusion

Congrats on implementing all the functions required for building a deep neural network!

We know it was a long assignment but going forward it will only get better. The next part of the assignment is easier.

In the next assignment you will put all these together to build two models:

- A two-layer neural network
- An L-layer neural network

You will in fact use these models to classify cat vs non-cat images!

1.4.7 Code of Deep Neural Network

```
import numpy as np
import h5py
import matplotlib.pyplot as plt
from testCases_v3 import *
from dnn_utils_v2 import sigmoid, sigmoid_backward, relu, relu_backward

#matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'


# GRADED FUNCTION: initialize_parameters
def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer

    Returns:
    parameters -- python dictionary containing your parameters:
                    W1 -- weight matrix of shape (n_h, n_x)
                    b1 -- bias vector of shape (n_h, 1)
                    W2 -- weight matrix of shape (n_y, n_h)
                    b2 -- bias vector of shape (n_y, 1)
    """

    np.random.seed(1)

    W1 = np.random.randn(n_h, n_x)*0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h)*0.01
    b2 = np.zeros((n_y, 1))

    assert(W1.shape == (n_h, n_x))
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

```
# GRADED FUNCTION: initialize_parameters_deep
def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each
↪   layer in our network

    Returns:
    parameters -- python dictionary containing your parameters "W1",
↪   "b1", ..., "WL", "bL":
                    Wl -- weight matrix of shape (layer_dims[l],
↪   layer_dims[l-1])
                    bl -- bias vector of shape (layer_dims[l], 1)
    """

    np.random.seed(3)
    parameters = {}
    L = len(layer_dims)            # number of layers in the network

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l],
        ↪  layer_dims[l-1]) * 0.01
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

        assert(parameters['W' + str(l)].shape == (layer_dims[l],
        ↪  layer_dims[l-1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))

    return parameters


# GRADED FUNCTION: linear_forward
def linear_forward(A, W, b):
    """
    Implement the linear part of a layer's forward propagation.

    Arguments:
    A -- activations from previous layer (or input data): (size of
↪   previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer,
↪   size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer,
↪   1)

    Returns:
    Z -- the input of the activation function, also called
↪   pre-activation parameter
```

```
    cache -- a python dictionary containing "A", "W" and "b" ; stored
↪   for computing the backward pass efficiently
    """

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

    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache


def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR->ACTIVATION layer

    Arguments:
    A_prev -- activations from previous layer (or input data): (size of
↪   previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer,
↪   size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer,
↪   1)
    activation -- the activation to be used in this layer, stored as a
↪   text string: "sigmoid" or "relu"

    Returns:
    A -- the output of the activation function, also called the
↪   post-activation value
    cache -- a python dictionary containing "linear_cache" and
↪   "activation_cache";
             stored for computing the backward pass efficiently
    """

    if activation == "sigmoid":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)

    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache
```

```
# GRADED FUNCTION: L_model_forward
def L_model_forward(X, parameters):
    """
    Implement forward propagation for the
↪   [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID computation

    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    parameters -- output of initialize_parameters_deep()

    Returns:
    AL -- last post-activation value
    caches -- list of caches containing:
                every cache of linear_relu_forward() (there are L-1 of
↪   them, indexed from 0 to L-2)
                the cache of linear_sigmoid_forward() (there is one,
↪   indexed L-1)
    """

    caches = []
    A = X
    L = len(parameters) // 2      # number of layers in the neural
    ↪  network

    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches"
    ↪  list.
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev, parameters['W' +
        ↪  str(l)], parameters['b' + str(l)], activation ="relu")
        caches.append(cache)

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    AL, cache = linear_activation_forward(A, parameters['W' + str(L)],
    ↪  parameters['b' + str(L)], activation = "sigmoid")
    caches.append(cache)

    assert(AL.shape == (1,X.shape[1]))

    return AL, caches


# GRADED FUNCTION: compute_cost
def compute_cost(AL, Y):
    """
    Implement the cost function defined by equation (7).
```

```
    Arguments:
    AL -- probability vector corresponding to your label predictions,
↪   shape (1, number of examples)
    Y -- true "label" vector (for example: containing 0 if non-cat, 1
↪   if cat), shape (1, number of examples)

    Returns:
    cost -- cross-entropy cost
    """

    m = Y.shape[1]
    # Compute loss from aL and y.
    cost = -(np.dot(Y,np.log(AL.T))+np.dot(1-Y,np.log(1-AL).T))/m

    cost = np.squeeze(cost)      # To make sure your cost's shape is
    ↪  what we expect (e.g. this turns [[17]] into 17).
    assert(cost.shape == ())

    return cost


# GRADED FUNCTION: linear_backward
def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single
↪   layer (layer l)

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of
↪   current layer l)
    cache -- tuple of values (A_prev, W, b) coming from the forward
↪   propagation in the current layer

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of
↪   the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l),
↪   same shape as W
    db -- Gradient of the cost with respect to b (current layer l),
↪   same shape as b
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = np.dot(dZ,A_prev.T)/m
    db = np.sum(dZ,axis=1,keepdims=True)/m
    dA_prev = np.dot(W.T,dZ)
```

```
    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db


# GRADED FUNCTION: linear_activation_backward

def linear_activation_backward(dA, cache, activation):
    """
    Implement the backward propagation for the LINEAR->ACTIVATION
↪   layer.

    Arguments:
    dA -- post-activation gradient for current layer l
    cache -- tuple of values (linear_cache, activation_cache) we store
↪   for computing backward propagation efficiently
    activation -- the activation to be used in this layer, stored as a
↪   text string: "sigmoid" or "relu"

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of
↪   the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l),
↪   same shape as W
    db -- Gradient of the cost with respect to b (current layer l),
↪   same shape as b
    """
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ =  relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    elif activation == "sigmoid":
        dZ =  sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db


# GRADED FUNCTION: L_model_backward
def L_model_backward(AL, Y, caches):
    """
    Implement the backward propagation for the [LINEAR->RELU] * (L-1)
↪   -> LINEAR -> SIGMOID group
```

```
    Arguments:
    AL -- probability vector, output of the forward propagation
↪   (L_model_forward())
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
    caches -- list of caches containing:
                every cache of linear_activation_forward() with "relu"
↪   (it's caches[l], for l in range(L-1) i.e l = 0...L-2)
                the cache of linear_activation_forward() with "sigmoid"
↪   (it's caches[L-1])

    Returns:
    grads -- A dictionary with the gradients
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as
    ↪ AL

    # Initializing the backpropagation
    dAL =  - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "AL, Y, caches".
    ↪ Outputs: "grads["dAL"], grads["dWL"], grads["dbL"]
    current_cache = caches[L-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] =
    ↪ linear_activation_backward(dAL, current_cache, "sigmoid")

    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        # Inputs: "grads["dA" + str(l + 2)], caches". Outputs:
        ↪ "grads["dA" + str(l + 1)] , grads["dW" + str(l + 1)] ,
        ↪ grads["db" + str(l + 1)]
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp =
        ↪ linear_activation_backward(grads["dA" + str(l+2)],
        ↪ current_cache, "relu")
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads

# GRADED FUNCTION: update_parameters
```

```
def update_parameters(parameters, grads, learning_rate):
    """
    Update parameters using gradient descent

    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients, output of
↪   L_model_backward

    Returns:
    parameters -- python dictionary containing your updated parameters
                  parameters["W" + str(l)] = ...
                  parameters["b" + str(l)] = ...
    """

    L = len(parameters) // 2 # number of layers in the neural network
    # Update rule for each parameter. Use a for loop.
    for l in range(L):
        parameters["W" + str(l+1)] =  parameters["W" + str(l+1)] -
        ↪  learning_rate * grads["dW" + str(l + 1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
        ↪  learning_rate * grads["db" + str(l + 1)]
    return parameters
```

1.5 Deep Neural Network for Image Classification: Application

Congratulations! Welcome to the fourth programming exercise of the deep learning specialization. You will now use everything you have learned to build a deep neural network that classifies **cat vs. non-cat images**.

In the second exercise, you used logistic regression to build cat vs. non-cat images and got a 68% accuracy. Your algorithm will now give you an 80% accuracy! you will see an improvement in accuracy relative to your previous logistic regression implementation. By completing this assignment, you will:

- Learn how to use all the helper functions you built in the previous assignment to build a model of any structure you want.
- Experiment with different model architectures and see how each one behaves.
- Recognize that it is always easier to build your helper functions before attempting to build a neural network from scratch.

This assignment prepares you well for the next course which dives deep into the techniques and strategies for parameters tuning and initializations. When you finish this, you will have finished the last programming assignment of Week 4, and also the last programming assignment of this course! Take your time to complete this assignment and make sure you get the expected outputs when working through the different exercises. In some code blocks, you will find a "#GRADED FUNCTION: functionName" comment. Please do not modify it. After you are done, submit your work and check your results. You need to score 70% to pass. Good luck :) !

After this assignment you will be able to: Build and apply a deep neural network to supervised learning. Let's get started!

1.5.1 Packages

Let's first import all the packages that you will need during this assignment.

- 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 Python.
- PIL and scipy are used here to test your model with your own picture at the end.
- dnn_app_utils provides the functions implemented in the "Building your Deep Neural Network: Step by Step" assignment to this notebook.

- np.random.seed(1) is used to keep all the random function calls consistent. It will help us grade your work.

```
import time
import numpy as np
import h5py
import matplotlib.pyplot as plt
import scipy
from PIL import Image
from scipy import ndimage
from dnn_app_utils_v2 import *

#matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

#load_ext autoreload
#autoreload 2

np.random.seed(1)
```

1.5.2 Dataset

You will use the same "Cat vs non-Cat" dataset as in "Logistic Regression as a Neural Network" (Assignment 2). The model you had built had 70% test accuracy on classifying cats vs non-cats images. Hopefully, your new model will perform a better!

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

- a training set of m_train images labelled as cat (1) or non-cat (0)
- a test set of m_test images labelled as cat and non-cat
- each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB).

Let's get more familiar with the dataset. Load the data by running the cell below.

```
train_x_orig, train_y, test_x_orig, test_y, classes = load_data()
```

The following code will show you an image in the dataset. Feel free to change the index and re-run the cell multiple times to see other images.

```
# Example of a picture
index = 10
plt.imshow(train_x_orig[index])
print ("y = " + str(train_y[0,index]) + ". It's a " +
↪  classes[train_y[0,index]].decode("utf-8") +  " picture.")
```

```
# Explore your dataset
m_train = train_x_orig.shape[0]
m_test =  test_x_orig.shape[0]
num_px = train_x_orig.shape[1]
```

As usual, you reshape and standardize the images before feeding them to the network. The code is given below.

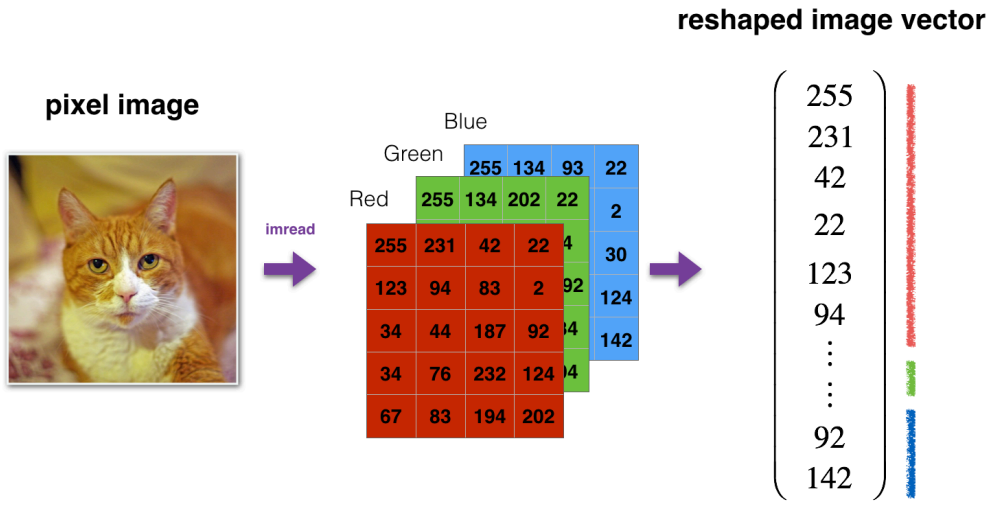


Figure 1.5.1 Image to vector conversion

12288 equals 64×64×3 which is the size of one reshaped image vector.

1.5.3 Architecture of your model

Now that you are familiar with the dataset, it is time to build a deep neural network to distinguish cat images from non-cat images.

You will build two different models:

- A 2-layer neural network
- An L-layer deep neural network

You will then compare the performance of these models, and also try out different values for L.

Let's look at the two architectures.

1.5.3.1 2-layer neural network

The model can be summarized as: *INPUT -> LINEAR -> RELU -> LINEAR -> SIGMOID -> OUTPUT*. Detailed Architecture of figure 1.5.2:

- The input is a (64,64,3) image which is flattened to a vector of size $(12288, 1)$.
- The corresponding vector: $[x_0, x_1, ..., x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ of size $(n^{[1]}, 12288)$.

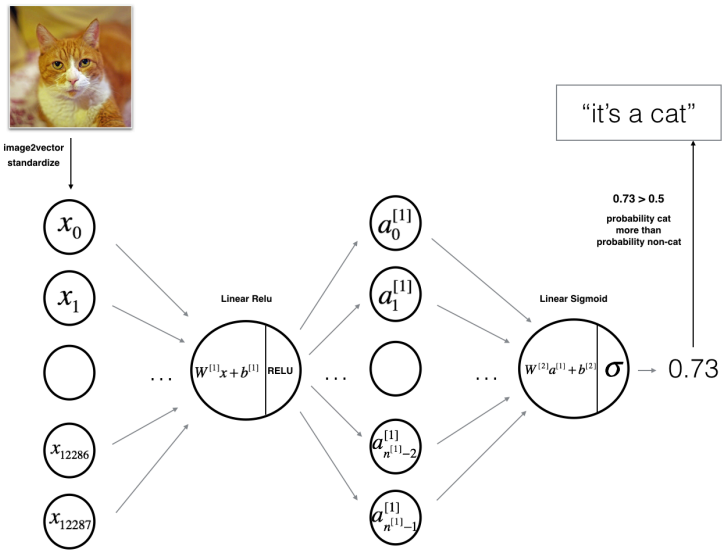


Figure 1.5.2 2-layer neural network

- You then add a bias term and take its relu to get the following vector: $[a_0^{[1]}, a_1^{[1]}, ..., a_{n^{[1]}-1}^{[1]}]^T$.
- You then repeat the same process.
- You multiply the resulting vector by $W^{[2]}$ and add your intercept (bias).
- Finally, you take the sigmoid of the result. If it is greater than 0.5, you classify it to be a cat.

1.5.3.2 L-layer deep neural network

It is hard to represent an L-layer deep neural network with the above representation. However, figure 1.5.3 is a simplified network representation:

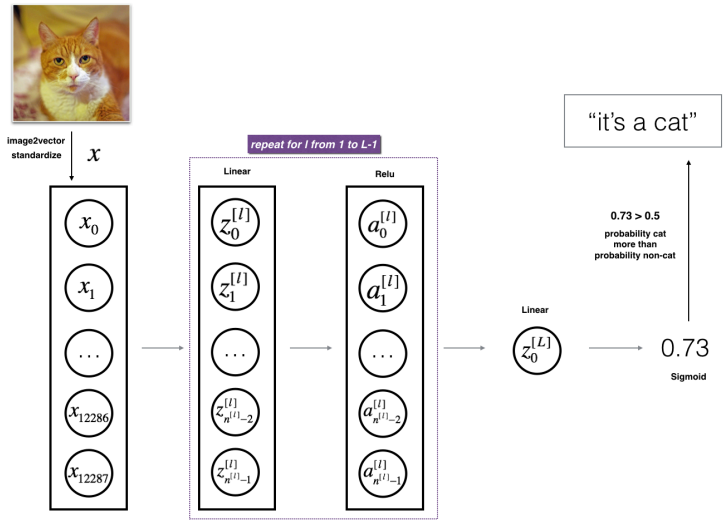


Figure 1.5.3 L-layer neural network

The model can be summarized as: *[LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID*. Detailed Architecture of figure 1.5.3:

- The input is a (64,64,3) image which is flattened to a vector of size (12288,1).
- The corresponding vector: $[x_0, x_1, ..., x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ and then you add the intercept $b^{[1]}$. The result is called the linear unit.
- Next, you take the relu of the linear unit. This process could be repeated several times for each $(W^{[l]}, b^{[l]})$ depending on the model architecture.
- Finally, you take the sigmoid of the final linear unit. If it is greater than 0.5, you classify it to be a cat.

1.5.3.3 General methodology

As usual you will follow the Deep Learning methodology to build the model:

1. Initialize parameters / Define hyperparameters
2. Loop for num_iterations:
    a. Forward propagation
    b. Compute cost function
    c. Backward propagation
    d. Update parameters (using parameters, and grads from backprop)
3. Use trained parameters to predict labels

Let's now implement those two models!

1.5.4 Two-layer neural network

Question: Use the helper functions you have implemented in the previous assignment to build a 2-layer neural network with the following structure: *LINEAR -> RELU -> LINEAR -> SIGMOID*. The functions you may need and their inputs are:

```
def initialize_parameters(n_x, n_h, n_y):
    ...
    return parameters
def linear_activation_forward(A_prev, W, b, activation):
    ...
    return A, cache
def compute_cost(AL, Y):
    ...
    return cost
def linear_activation_backward(dA, cache, activation):
    ...
    return dA_prev, dW, db
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

The whole code is as follows:

```
### CONSTANTS DEFINING THE MODEL ####
n_x = 12288     # num_px * num_px * 3
n_h = 7
n_y = 1
layers_dims = (n_x, n_h, n_y)


# GRADED FUNCTION: two_layer_model
def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075,
↪  num_iterations = 3000, print_cost=False):
    """
    Implements a two-layer neural network:
↪  LINEAR->RELU->LINEAR->SIGMOID.

    Arguments:
    X -- input data, of shape (n_x, number of examples)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of
↪  shape (1, number of examples)
    layers_dims -- dimensions of the layers (n_x, n_h, n_y)
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- If set to True, this will print the cost every 100
↪  iterations

    Returns:
    parameters -- a dictionary containing W1, W2, b1, and b2
    """

    np.random.seed(1)
    grads = {}
    costs = []                                      # to keep track of the cost
    m = X.shape[1]                                   # number of examples
    (n_x, n_h, n_y) = layers_dims

    # Initialize parameters dictionary, by calling one of the functions
    ↪  you'd previously implemented
    ### START CODE HERE ### ( 1 line of code)
    parameters =initialize_parameters(n_x, n_h, n_y)
    ### END CODE HERE ###

    # Get W1, b1, W2 and b2 from the dictionary parameters.
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Loop (gradient descent)
```

```
for i in range(0, num_iterations):

    # Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID.
    ↪  Inputs: "X, W1, b1". Output: "A1, cache1, A2, cache2".
    ### START CODE HERE ### ( 2 lines of code)
    A1, cache1 = linear_activation_forward(X, W1, b1, "relu")
    A2, cache2 = linear_activation_forward(A1, W2, b2, "sigmoid")
    ### END CODE HERE ###

    # Compute cost
    ### START CODE HERE ### ( 1 line of code)
    cost =  compute_cost(A2, Y)
    ### END CODE HERE ###

    # Initializing backward propagation
    dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))

    # Backward propagation. Inputs: "dA2, cache2, cache1". Outputs:
    ↪  "dA1, dW2, db2; also dA0 (not used), dW1, db1".
    ### START CODE HERE ### ( 2 lines of code)
    dA1, dW2, db2 = linear_activation_backward(dA2, cache2,
    ↪  "sigmoid")
    dA0, dW1, db1 = linear_activation_backward(dA1, cache1, "relu")
    ### END CODE HERE ###

    # Set grads['dWl'] to dW1, grads['db1'] to db1, grads['dW2'] to
    ↪  dW2, grads['db2'] to db2
    grads['dW1'] = dW1
    grads['db1'] = db1
    grads['dW2'] = dW2
    grads['db2'] = db2

    # Update parameters.
    ### START CODE HERE ### (approx. 1 line of code)
    parameters = update_parameters(parameters, grads,
    ↪  learning_rate)
    ### END CODE HERE ###

    # Retrieve W1, b1, W2, b2 from parameters
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Print the cost every 100 training example
    if print_cost and i % 100 == 0:
        print("Cost after iteration {}: {}".format(i,
        ↪  np.squeeze(cost)))
```

```
        if print_cost and i % 100 == 0:
            costs.append(cost)

    # plot the cost

    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

```
#input
parameters = two_layer_model(train_x, train_y, layers_dims = (n_x, n_h,
↪  n_y), num_iterations = 2500, print_cost=True)

#output
Cost after iteration 0: 0.693049735659989
Cost after iteration 100: 0.6464320953428849
Cost after iteration 200: 0.6325140647912678
......
Cost after iteration 2400: 0.048554785628770226
```

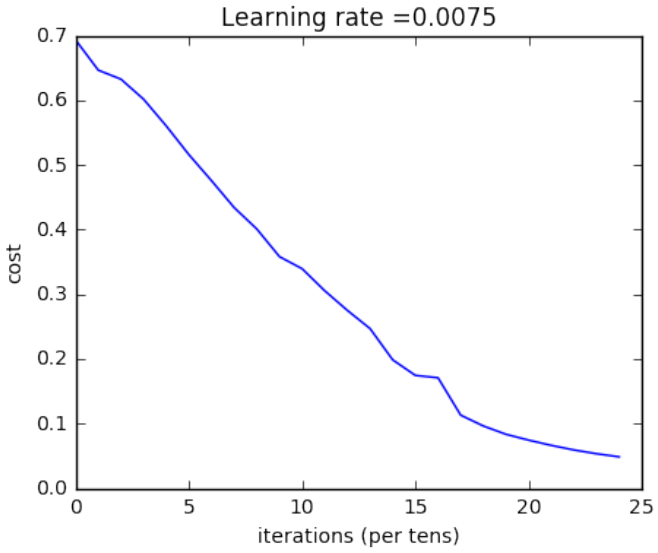


Figure 1.5.4 cost

Now, you can use the trained parameters to classify images from the dataset. To see your predictions on the training and test sets, run the code below.

```
predictions_train = predict(train_x, train_y, parameters)
#output
Accuracy: 1.0

predictions_test = predict(test_x, test_y, parameters)
#output
Accuracy: 0.72
```

Note: You may notice that running the model on fewer iterations (say 1500) gives better accuracy on the test set. This is called **"early stopping"** and we will talk about it in the next course. Early stopping is a way to prevent overfitting.

Congratulations! It seems that your 2-layer neural network has better performance (72%) than the logistic regression implementation (70%, assignment week 2). Let's see if you can do even better with an L-layer model.

1.5.5 L-layer Neural Network

Question: Use the helper functions you have implemented previously to build an L-layer neural network with the following structure: *[LINEAR -> RELU]×(L-1) -> LINEAR -> SIGMOID*. The functions you may need and their inputs are:

```
def initialize_parameters_deep(layer_dims):
    ...
    return parameters
def L_model_forward(X, parameters):
    ...
    return AL, caches
def compute_cost(AL, Y):
    ...
    return cost
def L_model_backward(AL, Y, caches):
    ...
    return grads
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

The whole code is as follows:

```
### CONSTANTS ###
layers_dims = [12288, 20, 7, 5, 1] #  5-layer model

# GRADED FUNCTION: L_layer_model
def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075,
↪  num_iterations = 3000, print_cost=False):#lr was 0.009
    """
    Implements a L-layer neural network:
↪   [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID.
```

```
    Arguments:
    X -- data, numpy array of shape (number of examples, num_px *
↪   num_px * 3)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of
↪   shape (1, number of examples)
    layers_dims -- list containing the input size and each layer size,
↪   of length (number of layers + 1).
    learning_rate -- learning rate of the gradient descent update rule
    num_iterations -- number of iterations of the optimization loop
    print_cost -- if True, it prints the cost every 100 steps

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

    np.random.seed(1)
    costs = []                         # keep track of cost

    # Parameters initialization.
    ### START CODE HERE ###
    parameters = initialize_parameters_deep(layers_dims)
    ### END CODE HERE ###

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

        # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR ->
        ↪  SIGMOID.
        ### START CODE HERE ### ( 1 line of code)
        AL, caches =  L_model_forward(X, parameters)
        ### END CODE HERE ###

        # Compute cost.
        ### START CODE HERE ### ( 1 line of code)
        cost =  compute_cost(AL, Y)
        ### END CODE HERE ###

        # Backward propagation.
        ### START CODE HERE ### ( 1 line of code)
        grads =  L_model_backward(AL, Y, caches)
        ### END CODE HERE ###

        # Update parameters.
        ### START CODE HERE ### ( 1 line of code)
        parameters =  update_parameters(parameters, grads,
        ↪  learning_rate)
        ### END CODE HERE ###
```

```
        # Print the cost every 100 training example
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))
        if print_cost and i % 100 == 0:
            costs.append(cost)

    # plot the cost
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

```
#input
parameters = L_layer_model(train_x, train_y, layers_dims,
↪  num_iterations = 2500, print_cost = True)

#output
Cost after iteration 0: 0.771749
Cost after iteration 100: 0.672053
Cost after iteration 200: 0.648263
......
Cost after iteration 2400: 0.092878
```

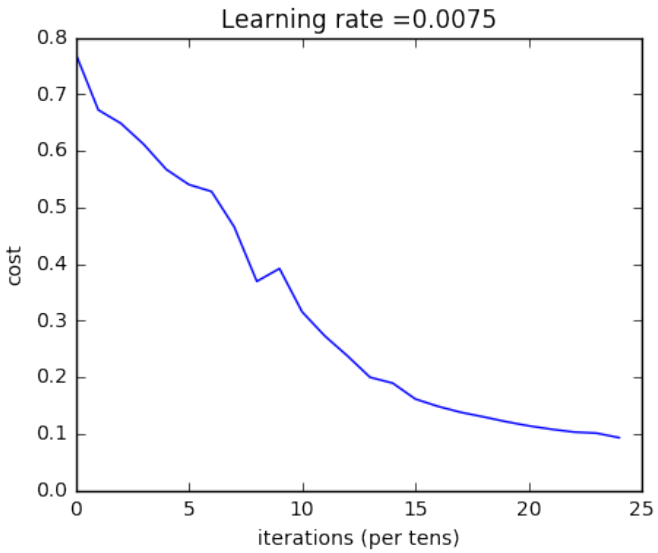


Figure 1.5.5 cost

Now, you can use the trained parameters to classify images from the dataset. To see your predictions on the training and test sets, run the code below.

```
pred_train = predict(train_x, train_y, parameters)
#output
Accuracy: 0.985645933014

pred_test = predict(test_x, test_y, parameters)
#output
Accuracy: 0.8
```

Congrats! It seems that your 5-layer neural network has better performance (80%) than your 2-layer neural network (72%) on the same test set.

This is good performance for this task. Nice job!

Though in the next course on "Improving deep neural networks" you will learn how to obtain even higher accuracy by systematically searching for better hyperparameters (learning_rate, layers_dims, num_iterations, and others you'll also learn in the next course).

1.5.6 Results Analysis

First, let's take a look at some images the L-layer model labeled incorrectly. This will show a few mislabeled images.

```
print_mislabeled_images(classes, test_x, test_y, pred_test)
```

Figure 1.5.6 mislabeled images

A few type of images the model tends to do poorly on include:

- Cat body in an unusual position
- Cat appears against a background of a similar color
- Unusual cat color and species
- Camera Angle
- Brightness of the picture
- Scale variation (cat is very large or small in image)

1.5.7 Test with your own image (optional/ungraded exercise)

Congratulations on finishing this assignment. You can use your own image and see the output of your model. To do that:

1. Click on “File” in the upper bar of this notebook, then click “Open” to go on your Coursera Hub.
2. Add your image to this Jupyter Notebook’s directory, in the “images” folder
3. Change your image’s name in the following code
4. Run the code and check if the algorithm is right (1 = cat, 0 = non-cat)!

```
## START CODE HERE ##
my_image = "test_cat3.jpg" # change this to the name of your image file
my_label_y = [1] # the true class of your image (1 -> cat, 0 ->
 ↪  non-cat)
## END CODE HERE ##

fname = "images/" + my_image
image = np.array(ndimage.imread(fname, flatten=False))
my_image = scipy.misc.imresize(image,
 ↪  size=(num_px,num_px)).reshape((num_px*num_px*3,1))
my_predicted_image = predict(my_image, my_label_y, parameters)

plt.imshow(image)
print ("y = " + str(np.squeeze(my_predicted_image)) + ", your L-layer
 ↪  model predicts a \"" +
 ↪  classes[int(np.squeeze(my_predicted_image)),].decode("utf-8") +
 ↪  "\" picture.")


#output:
Accuracy: 1.0
y = 1.0, your L-layer model predicts a "ca" picture.
```

1.5.8 Code of Deep Neural Network for Image Classification: Application

```
#matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

np.random.seed(1)

#Load the data
train_x_orig, train_y, test_x_orig, test_y, classes = load_data()

#============================================================
# #show an image in the dataset. Example of a picture
# index = 12
# plt.imshow(train_x_orig[index])
# print ("y = " + str(train_y[0,index]) + ". It's a " +
↪  classes[train_y[0,index]].decode("utf-8") +  " picture.")
#============================================================

# Explore your dataset
m_train = train_x_orig.shape[0]
m_test =  test_x_orig.shape[0]
num_px = train_x_orig.shape[1]

# Reshape the training and test examples
train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1).T   #
↪  The "-1" makes reshape flatten the remaining dimensions
test_x_flatten = test_x_orig.reshape(test_x_orig.shape[0], -1).T

# Standardize data to have feature values between 0 and 1.
train_x = train_x_flatten/255.
test_x = test_x_flatten/255.


### CONSTANTS DEFINING THE MODEL ####
n_x = 12288     # num_px * num_px * 3
n_h = 7
n_y = 1
layers_dims = (n_x, n_h, n_y)


# GRADED FUNCTION: two_layer_model

def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075,
↪  num_iterations = 3000, print_cost=False):
    """
    Implements a two-layer neural network:
↪  LINEAR->RELU->LINEAR->SIGMOID.
```

```
    Arguments:
    X -- input data, of shape (n_x, number of examples)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of
↪   shape (1, number of examples)
    layers_dims -- dimensions of the layers (n_x, n_h, n_y)
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- If set to True, this will print the cost every 100
↪   iterations

    Returns:
    parameters -- a dictionary containing W1, W2, b1, and b2
    """

    np.random.seed(1)
    grads = {}
    costs = []                                      # to keep track of the cost
    m = X.shape[1]                                  # number of examples
    (n_x, n_h, n_y) = layers_dims

    # Initialize parameters dictionary, by calling one of the functions
    ↪  you'd previously implemented
    parameters =initialize_parameters(n_x, n_h, n_y)

    # Get W1, b1, W2 and b2 from the dictionary parameters.
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Loop (gradient descent)
    for i in range(0, num_iterations):
        # Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID.
        ↪  Inputs: "X, W1, b1". Output: "A1, cache1, A2, cache2".
        A1, cache1 = linear_activation_forward(X, W1, b1, "relu")
        A2, cache2 = linear_activation_forward(A1, W2, b2, "sigmoid")

        # Compute cost
        cost =  compute_cost(A2, Y)

        # Initializing backward propagation
        dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))

        # Backward propagation. Inputs: "dA2, cache2, cache1". Outputs:
        ↪  "dA1, dW2, db2; also dA0 (not used), dW1, db1".
        dA1, dW2, db2 = linear_activation_backward(dA2, cache2,
        ↪  "sigmoid")
```

```
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, "relu")

        # Set grads['dWl'] to dW1, grads['db1'] to db1, grads['dW2'] to
        ↪  dW2, grads['db2'] to db2
        grads['dW1'] = dW1
        grads['db1'] = db1
        grads['dW2'] = dW2
        grads['db2'] = db2

        # Update parameters.
        parameters = update_parameters(parameters, grads,
        ↪  learning_rate)

        # Retrieve W1, b1, W2, b2 from parameters
        W1 = parameters["W1"]
        b1 = parameters["b1"]
        W2 = parameters["W2"]
        b2 = parameters["b2"]

        # Print the cost every 100 training example
        if print_cost and i % 100 == 0:
            print("Cost after iteration {}: {}".format(i,
            ↪  np.squeeze(cost)))
        if print_cost and i % 100 == 0:
            costs.append(cost)

    # plot the cost
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters



# GRADED FUNCTION: L_layer_model
def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075,
↪  num_iterations = 3000, print_cost=False):#lr was 0.009
    """
    Implements a L-layer neural network:
↪   [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID.

    Arguments:
    X -- data, numpy array of shape (number of examples, num_px *
↪   num_px * 3)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of
↪   shape (1, number of examples)
```

```
    layers_dims -- list containing the input size and each layer size,
↪   of length (number of layers + 1).
    learning_rate -- learning rate of the gradient descent update rule
    num_iterations -- number of iterations of the optimization loop
    print_cost -- if True, it prints the cost every 100 steps

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

    np.random.seed(1)
    costs = []                         # keep track of cost

    # Parameters initialization.
    parameters = initialize_parameters_deep(layers_dims)

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

        # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR ->
        ↪ SIGMOID.
        AL, caches =  L_model_forward(X, parameters)

        # Compute cost.
        cost =  compute_cost(AL, Y)

        # Backward propagation.
        grads =  L_model_backward(AL, Y, caches)

        # Update parameters.
        parameters =  update_parameters(parameters, grads,
        ↪ learning_rate)

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

    # plot the cost
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

```
### CONSTANTS ###
layers_dims = [12288, 20, 7, 5, 1] #  5-layer model

# two_layer_model
parameters_two_layer_model = two_layer_model(train_x, train_y,
↪  layers_dims = (n_x, n_h, n_y), num_iterations = 2500,
↪  print_cost=True)
predictions_train = predict(train_x, train_y,
↪  parameters_two_layer_model)
predictions_test = predict(test_x, test_y, parameters_two_layer_model)


# L_layer_model
parameters_L_layer_model = L_layer_model(train_x, train_y, layers_dims,
↪  num_iterations = 2500, print_cost = True)
pred_train = predict(train_x, train_y, parameters_L_layer_model)
pred_test = predict(test_x, test_y, parameters_L_layer_model)
```