Initialization

Training your neural network requires specifying an initial value of the weights. A well chosen initialization method will help learning a better model - depends on a lot of factors like the dataset for instance.

A well chosen initialization can:

- Speed up the convergence of gradient descent
- Increase the odds of gradient descent converging to a lower training (and generalization) error

To get started, run the following cell to load the packages and the planar dataset you will try to classify.

In [1]:

```
!git clone https://github.com/SanVik2000/EE5179-Final.git
```

```
Cloning into 'EE5179-Final'...
remote: Enumerating objects: 110, done.
remote: Counting objects: 100% (51/51), done.
remote: Compressing objects: 100% (46/46), done.
remote: Total 110 (delta 21), reused 4 (delta 1), pack-reused 59
Receiving objects: 100% (110/110), 7.57 MiB | 22.87 MiB/s, done.
Resolving deltas: 100% (42/42), done.
```

In [2]:

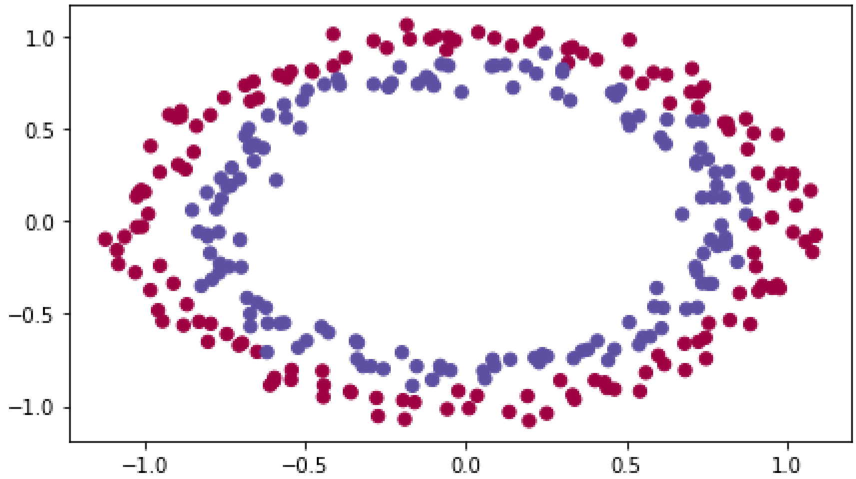
```
!cp /content/EE5179-Final/Tutorial-4/init_utils.py /content
```

In [3]:

```
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import sklearn.datasets
from init_utils import sigmoid, relu, compute_loss, forward_propagation, backwar
d_propagation
from init_utils import update_parameters, predict, load_dataset, plot_decision_b
oundary, predict_dec

%matplotlib inline
plt.rcParams['figure.figsize'] = (7.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# load image dataset: blue/red dots in circles
train_X, train_Y, test_X, test_Y = load_dataset()
```



You would like a classifier to separate the blue dots from the red dots.

1 - Neural Network model

You have been given a 3-layer neural network that has been already implemented for you. The model definition contains functions for the forward and backward passes and the optmization function as well (gradient decent in this case). Your task in this excercise is to initialise the weights of the model using the following three ways:

- *Zeros initialization* -- setting `initialization = "zeros"` in the input argument.
- *Random initialization* -- setting `initialization = "random"` in the input argument. This initializes the weights to large random values.
- *He initialization* -- setting `initialization = "he"` in the input argument. This initializes the weights to random values scaled according to a paper by He et al., 2015.

Instructions: Please quickly read over the code below, and run it. In the next part you will implement the three initialization methods that this `model()` calls.

In [4]:

```
def model(X, Y, learning_rate = 0.01, num_iterations = 15000, print_cost = True, 
initialization = "he"):
    """
    Implements a three-layer neural network: LINEAR->RELU->LINEAR->RELU->LINEAR-
>SIGMOID.

    Arguments:
    X -- input data, of shape (2, number of examples)
    Y -- true "label" vector (containing 0 for red dots; 1 for blue dots), of sh
ape (1, number of examples)
    learning_rate -- learning rate for gradient descent
    num_iterations -- number of iterations to run gradient descent
    print_cost -- if True, print the cost every 1000 iterations
    initialization -- flag to choose which initialization to use ("zeros","rando
m" or "he")

    Returns:
    parameters -- parameters learnt by the model
    """

    grads = {}
    costs = [] # to keep track of the loss
    m = X.shape[1] # number of examples
    layers_dims = [X.shape[0], 10, 5, 1]

    # Initialize parameters dictionary.
    if initialization == "zeros":
        parameters = initialize_parameters_zeros(layers_dims)
    elif initialization == "random":
        parameters = initialize_parameters_random(layers_dims)
    elif initialization == "he":
        parameters = initialize_parameters_he(layers_dims)

    # Loop (gradient descent)

    for i in range(0, num_iterations):

        # Forward propagation: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIG
MOID.
        a3, cache = forward_propagation(X, parameters)

        # Loss
        cost = compute_loss(a3, Y)

        # Backward propagation.
        grads = backward_propagation(X, Y, cache)

        # Update parameters.
        parameters = update_parameters(parameters, grads, learning_rate)

        # Print the loss every 1000 iterations
        if print_cost and i % 1000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
            costs.append(cost)

    # plot the loss
    plt.plot(costs)
    plt.ylabel('cost')
    plt.xlabel('iterations (per hundreds)')
```

```
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

2 - Zero initialization

There are two types of parameters to initialize in a neural network:

- the weight matrices $(W^{[1]}, W^{[2]}, W^{[3]}, \ldots, W^{[L-1]}, W^{[L]})$
- the bias vectors $(b^{[1]}, b^{[2]}, b^{[3]}, \ldots, b^{[L-1]}, b^{[L]})$

Exercise: Implement the following function to initialize all parameters to zeros. You'll see later that this does not work well since it fails to "break symmetry", but lets try it anyway and see what happens. Use np.zeros((..,..)) with the correct shapes.

In [5]:

```
# GRADED FUNCTION: initialize_parameters_zeros

def initialize_parameters_zeros(layers_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the size of each layer.

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ...,
 "WL", "bL":
                    W1 -- weight matrix of shape (layers_dims[1], layers_dims
[0])
                    b1 -- bias vector of shape (layers_dims[1], 1)
                    ...
                    WL -- weight matrix of shape (layers_dims[L], layers_dims[L-
1])
                    bL -- bias vector of shape (layers_dims[L], 1)
    """

    parameters = {}
    L = len(layers_dims)

    for l in range(1, L):
        ### START CODE HERE ### (≈ 2 lines of code)

        parameters["W"+str(l)]=np.zeros((layers_dims[l],layers_dims[l-1]))
        parameters["b"+str(l)]=np.zeros((layers_dims[l],1))

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

        ### END CODE HERE ###
    return parameters
```

In [6]:

```
parameters = initialize_parameters_zeros([3,2,1])
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[0. 0. 0.]
 [0. 0. 0.]]
b1 = [[0.]
 [0.]]
W2 = [[0. 0.]]
b2 = [[0.]]
```

Expected Output:

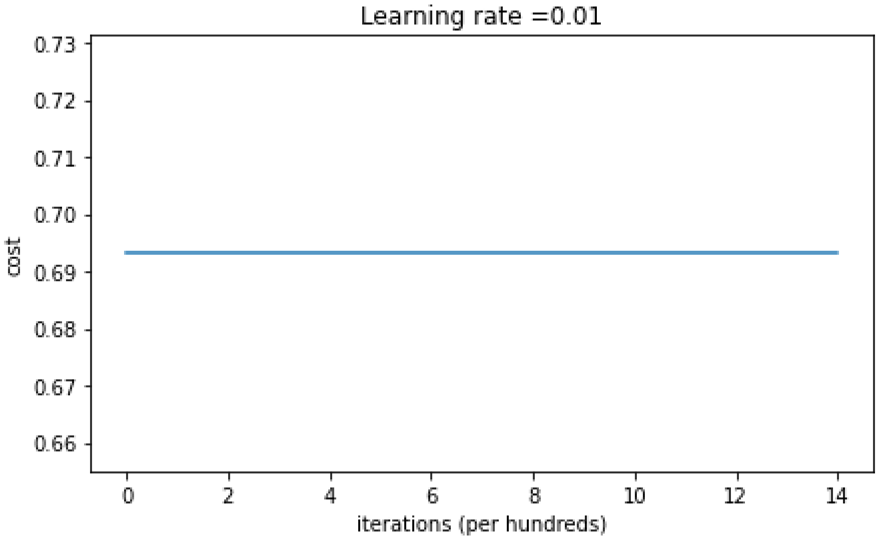
W1	[[0. 0. 0.] [0. 0. 0.]]
b1	[[0.] [0.]]
W2	[[0. 0.]]
b2	[[0.]]

Run the following code to train your model on 15,000 iterations using zeros initialization.

In [7]:

```
parameters = model(train_X, train_Y, initialization = "zeros")
print ("On the train set:")
predictions_train = predict(train_X, train_Y, parameters)
print ("On the test set:")
predictions_test = predict(test_X, test_Y, parameters)
```

```
Cost after iteration 0: 0.6931471805599453
Cost after iteration 1000: 0.6931471805599453
Cost after iteration 2000: 0.6931471805599453
Cost after iteration 3000: 0.6931471805599453
Cost after iteration 4000: 0.6931471805599453
Cost after iteration 5000: 0.6931471805599453
Cost after iteration 6000: 0.6931471805599453
Cost after iteration 7000: 0.6931471805599453
Cost after iteration 8000: 0.6931471805599453
Cost after iteration 9000: 0.6931471805599453
Cost after iteration 10000: 0.6931471805599455
Cost after iteration 11000: 0.6931471805599453
Cost after iteration 12000: 0.6931471805599453
Cost after iteration 13000: 0.6931471805599453
Cost after iteration 14000: 0.6931471805599453
```




```
On the train set:
Accuracy: 0.5
On the test set:
Accuracy: 0.5
```

The performance is really bad, and the cost does not really decrease, and the algorithm performs no better than random guessing. Why? Lets look at the details of the predictions and the decision boundary:

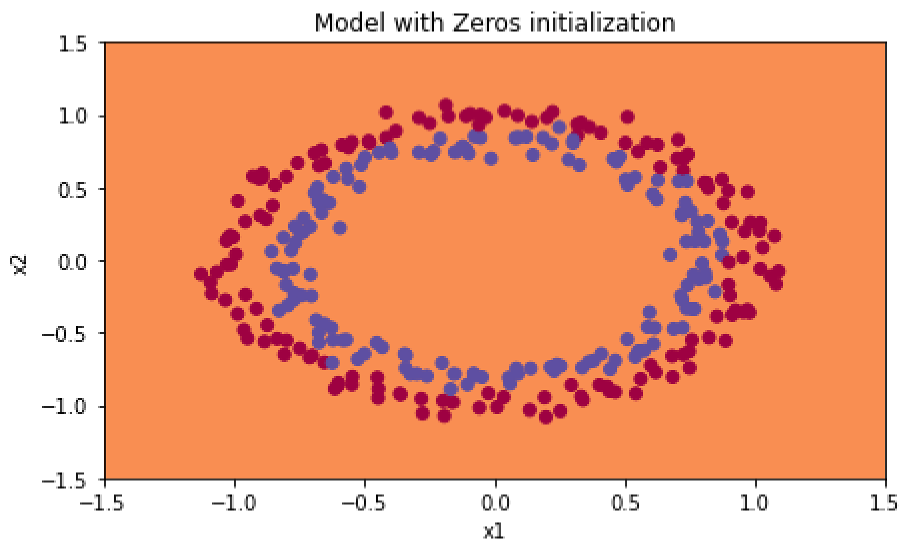
In [8]:

```
print ("predictions_train = " + str(predictions_train))
print ("predictions_test = " + str(predictions_test))
```

```
predictions_train = [[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 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 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
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 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 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 0 0 0
0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0]]
predictions_test = [[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 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 0 0 0 0 0 0]]
```

In [9]:

```
plt.title("Model with Zeros initialization")
axes = plt.gca()
axes.set_xlim([-1.5,1.5])
axes.set_ylim([-1.5,1.5])
plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X, train_Y)
```



The model is predicting 0 for every example. Generally, when all the weights of the model are initalized to zero, the network fails to break symmetry...

In other words, every neuron in our model tends to learn the same thing.

3 - Random initialization

To break symmetry, lets intialize the weights randomly. Following random initialization, each neuron can then proceed to learn a different function of its inputs. In this exercise, you will see what happens if the weights are intialized randomly, but to very large values.

Exercise: Implement the following function to initialize your weights to large random values (scaled by *10) and your biases to zeros. Use `np.random.randn(..,..) * 10` for weights and `np.zeros((.., ..))` for biases. We are using a fixed `np.random.seed(..)` to make sure your "random" weights match ours, so don't worry if running several times your code gives you always the same initial values for the parameters.

In [10]:

```
# GRADED FUNCTION: initialize_parameters_random

def initialize_parameters_random(layers_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the size of each layer.

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ...,
 "WL", "bL":
                    W1 -- weight matrix of shape (layers_dims[1], layers_dims
[0])
                    b1 -- bias vector of shape (layers_dims[1], 1)
                    ...
                    WL -- weight matrix of shape (layers_dims[L], layers_dims[L-
1])
                    bL -- bias vector of shape (layers_dims[L], 1)
    """

    np.random.seed(3)               # This seed makes sure your "random" numbers
will be the as ours
    parameters = {}
    L = len(layers_dims)            # integer representing the number of layers

    for l in range(1, L):
        ### START CODE HERE ### (≈ 2 lines of code)
        parameters["W"+str(l)]=np.random.randn(layers_dims[l],layers_dims[l-1])*
10
        parameters["b"+str(l)]=np.zeros((layers_dims[l],1))

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

        ### END CODE HERE ###

    return parameters
```

In [11]:

```
parameters = initialize_parameters_random([3, 2, 1])
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[ 17.88628473   4.36509851   0.96497468]
 [-18.63492703  -2.77388203  -3.54758979]]
b1 = [[0.]
 [0.]]
W2 = [[-0.82741481 -6.27000677]]
b2 = [[0.]]
```

Expected Output:

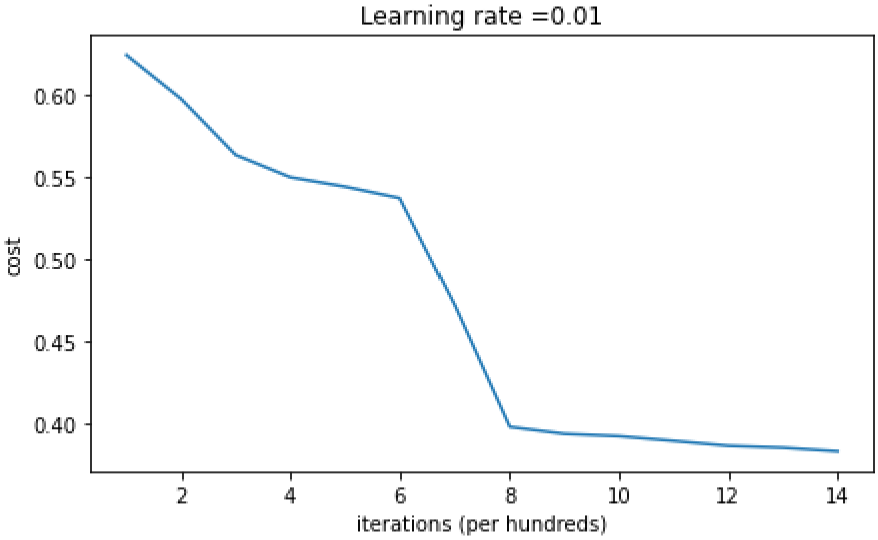
W1	[[17.88628473 4.36509851 0.96497468] [-18.63492703 -2.77388203 -3.54758979]]
b1	[[0.] [0.]]
W2	[[-0.82741481 -6.27000677]]
b2	[[0.]]

Run the following code to train your model on 15,000 iterations using random initialization.

In [19]:

```
parameters = model(train_X, train_Y, initialization = "random")
print ("On the train set:")
predictions_train = predict(train_X, train_Y, parameters)
print ("On the test set:")
predictions_test = predict(test_X, test_Y, parameters)
```

```
Cost after iteration 0: inf
Cost after iteration 1000: 0.6247924745506072
Cost after iteration 2000: 0.5980258056061102
Cost after iteration 3000: 0.5637539062842213
Cost after iteration 4000: 0.5501256393526495
Cost after iteration 5000: 0.5443826306793814
Cost after iteration 6000: 0.5373895855049121
Cost after iteration 7000: 0.47157999220550006
Cost after iteration 8000: 0.39770475516243037
Cost after iteration 9000: 0.3934560146692851
Cost after iteration 10000: 0.3920227137490125
Cost after iteration 11000: 0.38913700035966736
Cost after iteration 12000: 0.3861358766546214
Cost after iteration 13000: 0.38497629552893475
Cost after iteration 14000: 0.38276694641706693
```




```
On the train set:
Accuracy: 0.83
On the test set:
Accuracy: 0.86
```

If you see "inf" as the cost after the iteration 0, this is because of numerical roundoff; a more numerically sophisticated implementation would fix this. But this isn't worth worrying about for our purposes.

Anyway, it looks like you have broken symmetry, and this gives better results. than before. The model is no longer outputting all 0s.

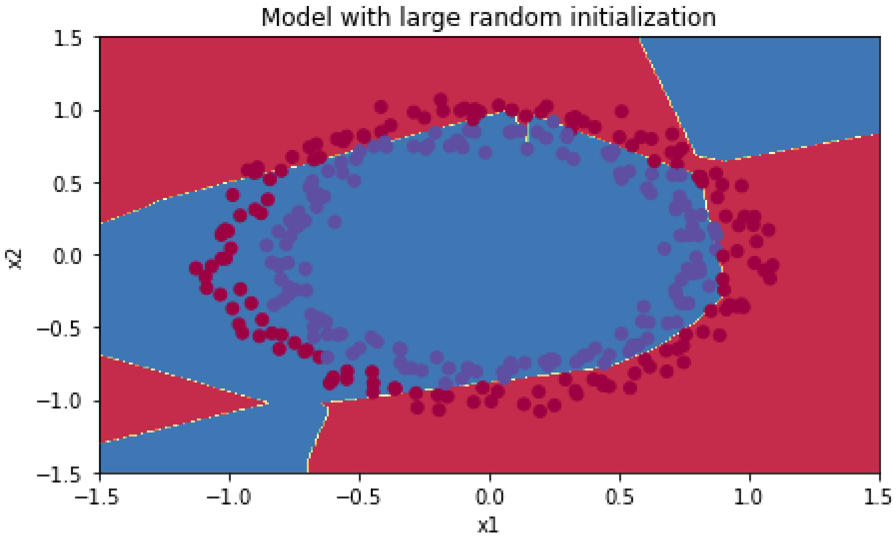
In [13]:

```
print (predictions_train)
print (predictions_test)
```

```
[[1 0 1 1 0 0 1 1 1 1 1 0 1 0 0 1 0 1 1 0 0 0 1 0 1 1 1 1 1 1 0 1 1
0 0 1
  1 1 1 1 1 1 1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 1 1 1 1 0 1 1 0 1 0 1 1
1 1 0
  0 0 0 0 1 0 1 0 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1 0 1 1 0 1 0 1 1 0
1 1 0
  1 0 1 1 0 0 1 0 0 1 1 0 1 1 1 0 1 0 0 1 0 1 1 1 1 1 1 1 0 1 1 0 0
1 1 0
  0 0 1 0 1 0 1 0 1 1 1 0 0 1 1 1 1 0 1 1 0 1 0 1 1 0 1 0 1 1 1 1 0
1 1 1
  1 0 1 0 1 0 1 1 1 1 0 1 1 0 1 1 0 1 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1
0 0 1
  0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 1 1 0 1 0 0 1 1 0 1 1 1 0 0 0 1 1
0 1 1
  1 1 0 1 1 0 1 1 1 0 0 1 0 0 0 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0
1 1 1
  1 1 1 1 0 0 0 1 1 1 1 0]]
[[1 1 1 1 0 1 0 1 1 0 1 1 1 0 0 0 0 1 0 1 0 0 1 0 1 0 1 1 1 1 1 0 0
0 0 1
  0 1 1 0 0 1 1 1 1 1 0 1 1 1 0 1 0 1 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1
0 1 0
  1 1 1 1 1 0 1 0 0 1 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0]]
```

In [14]:

```
plt.title("Model with large random initialization")
axes = plt.gca()
axes.set_xlim([-1.5,1.5])
axes.set_ylim([-1.5,1.5])
plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X, train_Y)
```



Excercise 1

Mention your Observations: (Cost Function Curve, How initialization improves, training time etc..)

- Observation-1: Error decreases fastly
- Observation-2: The decision boundaries are somewhat better but can be improved.
- Observation-3: trains faster

4 - He initialization

Finally, let us now implement "He Initialization"; this is named for the first author of He et al., 2015. (This is similar to "Xavier initialization" where weights are initialized randomly and scaled as follows $W^{[l]}$ of `sqrt(1./layers_dims[l-1])` where He initialization scale to `sqrt(2./layers_dims[l-1])` .)

Exercise: Implement the following function to initialize your parameters with He initialization.

Hint: This function is similar to the previous `initialize_parameters_random(...)` . The only difference is that instead of multiplying `np.random.randn(..,..)` by 10, you will multiply it by $\sqrt{\frac{2}{\text{dimension of the previous layer}}}$, which is what He initialization recommends for layers with a ReLU activation.

In [15]:

```
# GRADED FUNCTION: initialize_parameters_he

def initialize_parameters_he(layers_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the size of each layer.

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ...,
 "WL", "bL":
                    W1 -- weight matrix of shape (layers_dims[1], layers_dims
[0])
                    b1 -- bias vector of shape (layers_dims[1], 1)
                    ...
                    WL -- weight matrix of shape (layers_dims[L], layers_dims[L-
1])
                    bL -- bias vector of shape (layers_dims[L], 1)
    """

    np.random.seed(3)
    parameters = {}
    L = len(layers_dims) - 1 # integer representing the number of layers
    import math
    for l in range(1, L + 1):
        ### START CODE HERE ### (≈ 2 lines of code)
        parameters["W"+str(l)]=np.random.randn(layers_dims[l],layers_dims[l-1])*
np.sqrt(2/layers_dims[l-1])
        parameters["b"+str(l)]=np.zeros((layers_dims[l],1))

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

        ### END CODE HERE ###

    return parameters
```

In [16]:

```
parameters = initialize_parameters_he([2, 4, 1])
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[ 1.78862847  0.43650985]
 [ 0.09649747 -1.8634927 ]
 [-0.2773882  -0.35475898]
 [-0.08274148 -0.62700068]]
b1 = [[0.]
 [0.]
 [0.]
 [0.]]
W2 = [[-0.03098412 -0.33744411 -0.92904268  0.62552248]]
b2 = [[0.]]
```

Expected Output:

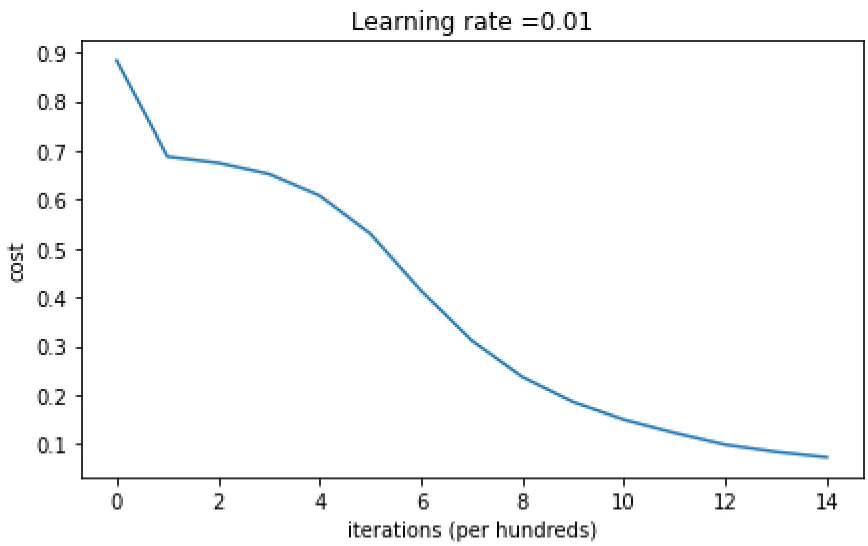
W1	[[1.78862847 0.43650985] [0.09649747 -1.8634927] [-0.2773882 -0.35475898] [-0.08274148 -0.62700068]]
b1	[[0.] [0.] [0.] [0.]]
W2	[[-0.03098412 -0.33744411 -0.92904268 0.62552248]]
b2	[[0.]]

Run the following code to train your model on 15,000 iterations using He initialization.

In [20]:

```
parameters = model(train_X, train_Y, initialization = "he")
print ("On the train set:")
predictions_train = predict(train_X, train_Y, parameters)
print ("On the test set:")
predictions_test = predict(test_X, test_Y, parameters)
```

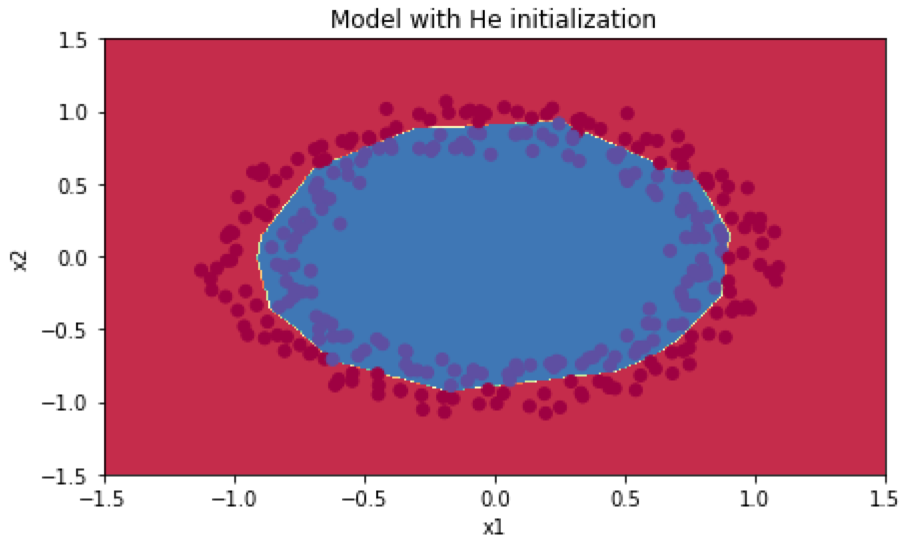
```
Cost after iteration 0: 0.8830537463419761
Cost after iteration 1000: 0.6879825919728063
Cost after iteration 2000: 0.6751286264523371
Cost after iteration 3000: 0.6526117768893805
Cost after iteration 4000: 0.6082958970572938
Cost after iteration 5000: 0.5304944491717495
Cost after iteration 6000: 0.4138645817071794
Cost after iteration 7000: 0.3117803464844441
Cost after iteration 8000: 0.23696215330322562
Cost after iteration 9000: 0.1859728720920684
Cost after iteration 10000: 0.15015556280371808
Cost after iteration 11000: 0.12325079292273551
Cost after iteration 12000: 0.09917746546525937
Cost after iteration 13000: 0.08457055954024283
Cost after iteration 14000: 0.07357895962677366
```




```
On the train set:
Accuracy: 0.9933333333333333
On the test set:
Accuracy: 0.96
```

In [18]:

```
plt.title("Model with He initialization")
axes = plt.gca()
axes.set_xlim([-1.5,1.5])
axes.set_ylim([-1.5,1.5])
plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X, train_Y)
```



Excercise 2

Mention your Observations: (Cost Function Curve, How initialization improves, training time etc..)

- Observation-1: Cost decreases based on gradients i.e. initially slow then fast
- Observation-2: Better decision boundary. Model learns properly.
- Observation-3: Training time is slightly more than that of random initialisation

5 - Conclusions

Here are the results of our three models with same structure (network and training iterations):

Model	**Train Accuracy**	**Problem/Comment**
3-layer NN without Zero-ntialization	50%	Fails to break symmetry
3-layer NN with Random-Initalization	83%	Too Large weights
3-layer NN with He-Initialization	99%	Recommended Method