Abstract

Target image recognition is the technique in which we want to input an image to a classifier model and the output tells whether the input image is the target image or not. We want to train our classifier model over a large set of pair of input image and corresponding label. If we train our classifier model over enough dataset, we can expect our classifier model to classify the input image correctly with the help of learnt parameters. The target image in this project is the image of animal cat. The dataset used in the project is a labelled dataset containing image information and the label of the image. Thus, the learning mechanism is called supervised learning. The classifier model in our project is a Neural Network model to recognize images of animal cats than that of non-cats. The learning algorithm used in the project is a logistic regression which is a binary classifier, i.e. it can classify the input data into one of two classes. In my approach, I have modelled a Four-Layer-Deep Neural Network to learn features of images over the training dataset for a number of iterations and I have tested my Neural Network model on the test dataset achieving an accuracy of 80%.

Keywords: Target image recognition, Classifier model, Supervised learning, Neural network, Logistic regression.

Introduction

What is Neural Network?

Neural Network is a network of neurons connected together in order to achieve complex tasks such as image recognition, voice recognition, object detection and so on. Neurons are basically the building blocks of the Neural Networks. They take input from the data, compute some function and provide final output or output to the neurons in the next layer. Many neurons can be stacked together in several layers in order to perform more complex tasks. Such a Neural Network is known as a Deep Neural Network. The process of training a Deep Neural Network is known as Deep Learning. Neural Networks, when trained on a large number of input data and output label pair, are extremely good at coming up with a mapping function which can map input data to the output labels.

Parameters of Neural Network

Each neuron in the Neural Network is associated with some parameters, namely weight and bias. These parameters are initially randomly initialized. Using some learning algorithm and some cost function corresponding to it, we can update the parameters such that the cost function is minimized with every iteration of our training process, and at the end of our training, we have our learnt parameters using which we can classify the input data as target data or non-target data. We train our Neural Network for sufficiently large number of iterations until our cost function is minimized. The learning algorithm used in the project is the logistic regression learning algorithm, the cost function used is the logistic regression cost function and the data is the image.

1. How images are stored in computer?

Images are stored in computer as three-dimensional matrices. Each dimension corresponds to one of the three colour channels – Red, Green and Blue, called as RGB channels. Each channel is a two-dimensional matrix of dimensions equal to width and height of the image in pixels. The entry in each matrix cell is the pixel intensity value of that pixel corresponding to the colour channel, ranging from 0 to 255.

2. Input feature vector X

We convert the three-dimensional image matrix into a single column vector of dimension [(height * width * 3), 1] where height and width are image height and width in pixels. For a number of training set images, these column vectors are stacked together column-wise making the input feature matrix X of dimension [nx, m] where "nx" is the dimension of each image feature vector and "m" is the number of training set images.

3. Standardizing the dataset

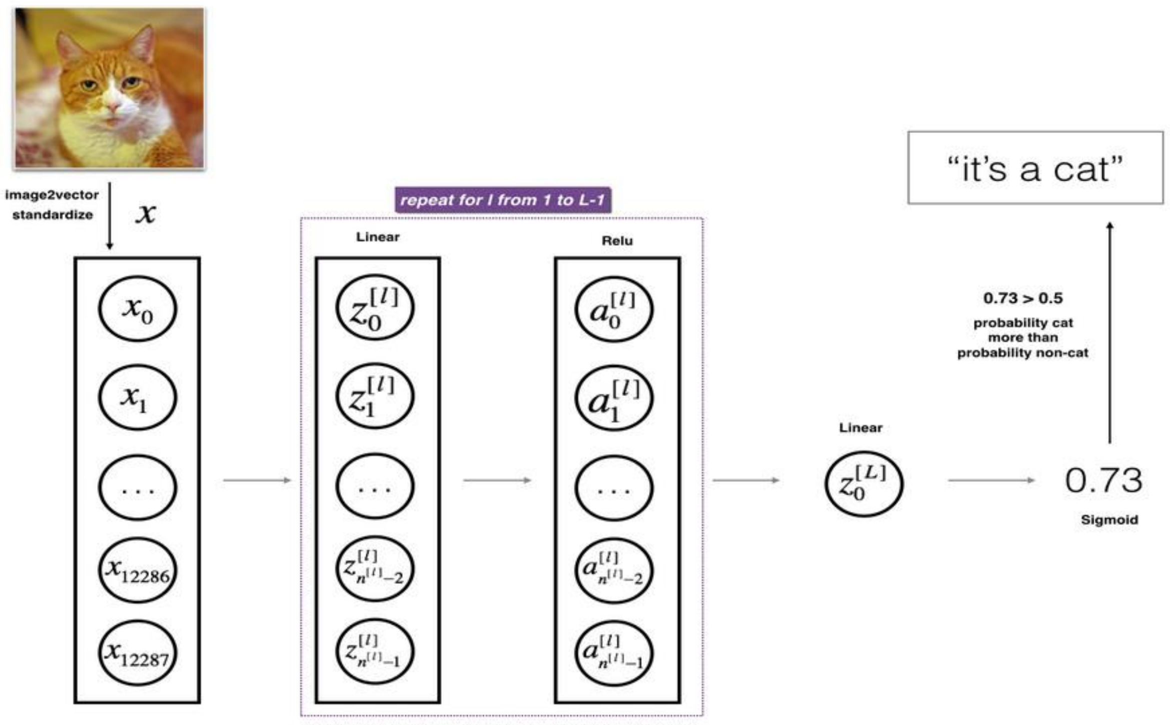
Standardizing the features around the center and 0 with a standard deviation of 1 is important when we compare measurements that have different units. Variables that are measured at different scales do not contribute equally to the analysis and might end up creating a bias. For example, A variable that ranges between 0 and 1000 will outweigh a variable that ranges between 0 and 1. Using these variables without standardization will give the variable with the larger range weight of 1000 in the analysis. Transforming the data to comparable scales can prevent this problem. Typical data standardization procedures equalize the range and/or data variability.

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

4. Hyperparameters

Hyperparameters are the parameters which influence the value of our model parameters. These include – learning rate (α), number of iterations for which we want to train our Neural Network model, number of hidden layers in our Neural Network, number of hidden units in a particular layer, choice of activation function and so on.

5. Architecture of the model



- 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, we 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, we take the sigmoid of the final linear unit. If it is greater than 0.5, we classify it to be a cat.

6. Outline of the project

To build our neural network, we will be implementing several "helper functions". These helper functions will be used to build an L-layer neural network. Outline :

- Initialize the parameters for a two-layer network and for an L-layer neural network.

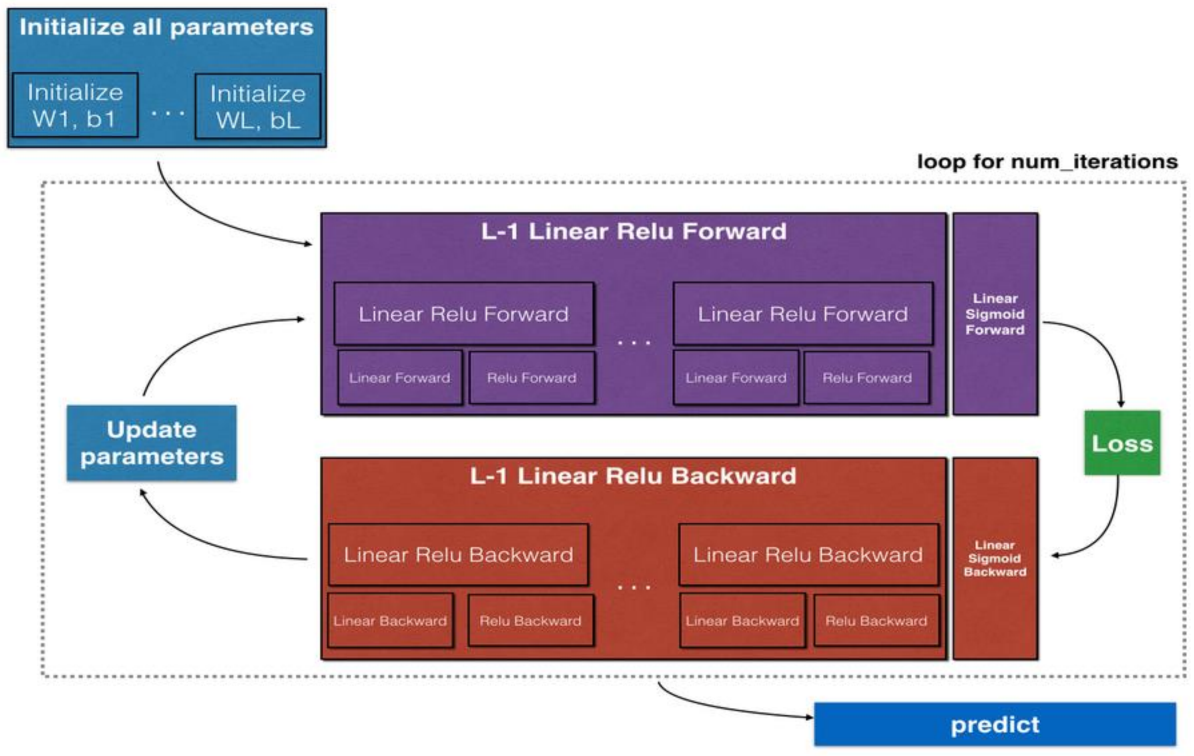
- Implement the forward propagation module (shown in purple in the figure below).
  - Complete the LINEAR part of a layer's forward propagation step (resulting in $Z^{[l]}$).

    - Given is 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 times (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 ACTIVATION 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.



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.

7. General methodology

- Initialize parameters / define hyperparameters.
- Loop for number of iterations:
    - Forward propagation
    - Compute cost function
    - Backward propagation
    - Update parameters (using parameters, and grads from backprop)
- Use trained parameters to predict labels.

7.1. Initializing parameters

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 parameters initialization, we should make sure that our 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:

$$W = \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} \quad X = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad b = \begin{bmatrix} s \\ t \\ u \end{bmatrix} \tag{2}$$

Then $WX + b$ will be:

$$WX + b = \begin{bmatrix} (ja + kd + lg) + s & (jb + ke + lh) + s & (jc + kf + li) + s \\ (ma + nd + og) + t & (mb + ne + oh) + t & (mc + nf + oi) + t \\ (pa + qd + rg) + u & (pb + qe + rh) + u & (pc + qf + ri) + u \end{bmatrix} \tag{3}$$

- 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.randn(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.
- Here is the implementation for L=1 (one-layer neural network). It should inspire us 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))
```

7.2. Forward Propagation module

7.2.1. Linear Forward

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

- LINEAR
- LINEAR -> ACTIVATION where ACTIVATION will be either ReLU or Sigmoid.
- [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID (whole model)

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

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

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

7.2.2. Linear-Activation Forward

In this project, we will use two activation functions:

- **Sigmoid**: $\sigma(Z) = \sigma(WA + b) = \frac{1}{1+e^{-(WA+b)}}$. We have been provided 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 we could just call:

  A, activation_cache = sigmoid(Z)

- **ReLU**: The mathematical formula for ReLu is A = RELU(Z) = max(0,Z). We have been provided 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 we could just call:

  A, activation_cache = relu(Z)

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

We will implement the forward propagation of the LINEAR->ACTIVATION layer. Mathematical relation is:

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

where the activation "g" can be sigmoid() or relu(). Use linear_forward() and the correct activation function.

7.2.3. L-model Forward

For even more convenience when implementing the L-layer Neural Net, we 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.

In the code below, the variable AL will denote

$$\mathbf{A^{[L]} = \sigma(Z^{[L]}) = \sigma(W^{[L]}A^{[L-1]} + b^{[L]}).}$$

(This is sometimes also called Yhat, i.e., this is $\hat{Y}$.)

- Use the functions we had previously written
- Use a for loop to replicate [LINEAR->RELU] (L-1) times
- Keep track of the caches in the "caches" list.

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

7.3. Cost function

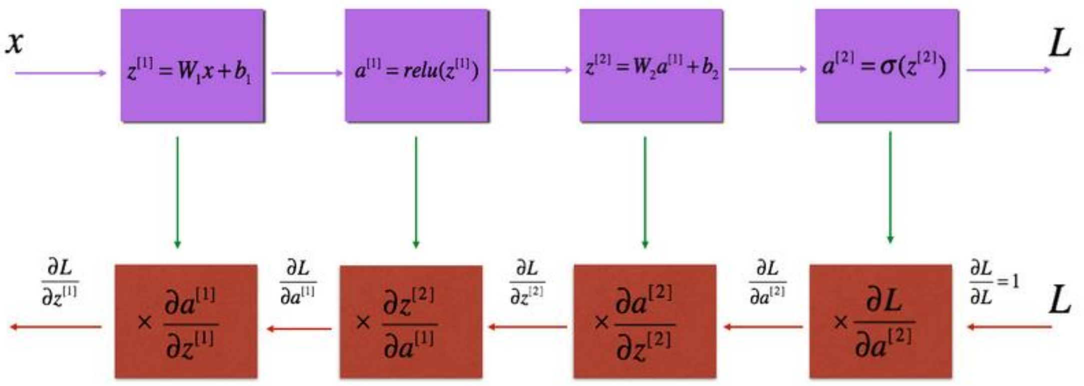
We need to compute the cost, because we want to check how well our model is actually performing. Later we would aim to minimize this cost function over number of iterations by going in the direction of global minimum or global optimum using the gradient descent algorithm.

We compute the cross-entropy cost *J*, using the following formula:

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

7.4. Backward Propagation module

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



x
$z^{[1]} = W_1 x + b_1$
$a^{[1]} = relu(z^{[1]})$
$z^{[2]} = W_2 a^{[1]} + b_2$
$a^{[2]} = \sigma(z^{[2]})$
L
$\frac{\partial L}{\partial z^{[1]}}$
$\times \frac{\partial a^{[1]}}{\partial z^{[1]}}$
$\frac{\partial L}{\partial a^{[1]}}$
$\times \frac{\partial z^{[2]}}{\partial a^{[1]}}$
$\frac{\partial L}{\partial z^{[2]}}$
$\times \frac{\partial a^{[2]}}{\partial z^{[2]}}$
$\frac{\partial L}{\partial a^{[2]}}$
$\times \frac{\partial L}{\partial a^{[2]}}$
$\frac{\partial L}{\partial L} = 1$
L

Figure 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, we are going to build the backward propagation in three steps:

- LINEAR backward
- LINEAR -> ACTIVATION backward where ACTIVATION computes the derivative of either the ReLU or sigmoid activation.
- [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID backward (whole model).

7.4.1. Linear Backward

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

Suppose we have already calculated the derivative $dZ^{[l]} = \frac{\partial L}{\partial Z^{[l]}}$. Now we want to get $(dW^{[l]}, db^{[l]}, dA^{[l-1]})$. These three outputs are computed using the input $dZ^{[l]}$. Here are the formulas you need:

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

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

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

7.4.2. Linear Activation Backward

Next, we will create a function that merges the two helper functions:

- linear_backward
- the backward step for the activation linear_activation_backward.

To help us implement linear_activation_backward, we are provided with 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

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

7.4.3. L-model Backward

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

Initializing backpropagation: To backpropagate through this network, we know that the output is, $A^{[L]} = \sigma(Z^{[L]})$. Our code thus needs to compute **dAL =** $\frac{\partial L}{\partial A^{[L]}}$.

To do so, we use this formula derived using calculus:

dAL = - $\left(\frac{Y}{AL} - \frac{(1-Y)}{(1-AL)} \right)$ (derivative of cost with respect to AL)

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

$$grads["dW" + str(l)] = dW^{[l]}$$

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

7.5. Update Parameters

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

$$W^{[l]} = W^{[l]} - \alpha \, dW^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]}$$

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

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

7.6. Predict labels

After training the model we get the learnt parameters **w** and **b.** We are able to use these parameters to predict the labels for a test dataset **X'**. We implement the predict() function. There are 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.

Using the predicted labels and the true labels we can calculate the accuracy of our Neural Network model.

8. Result and analysis

Number of training examples: 209
Number of testing examples: 50
Each image is of size: (64, 64, 3)
train_x_orig shape: (209, 64, 64, 3)

train_y shape: (1, 209)
test_x_orig shape: (50, 64, 64, 3)
test_y shape: (1, 50)

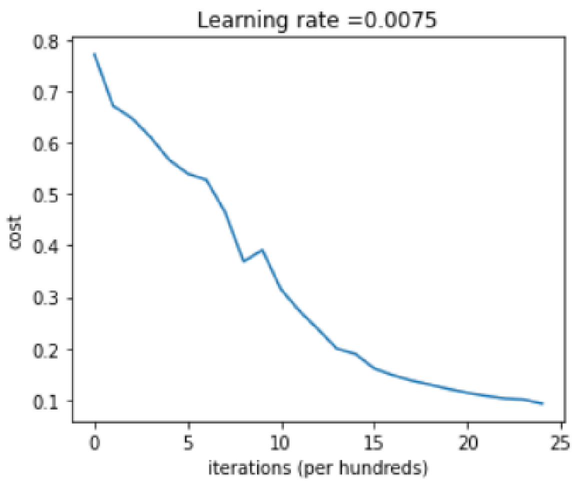
After reshaping the and conversion of images into feature vectors

train_x's shape: (12288, 209)
test_x's shape: (12288, 50)

Hyperparameters :

- Layer dimensions : [12288, 20, 7, 5, 1], i.e.
  - Total number of layers = 4
  - Number of input features per image = 12288 (i.e. 3 x 64 x 64)
  - Number of hidden layers = 3
  - Number of neurons in the first hidden layer = 20
  - Number of neurons in the second hidden layer = 7
  - Number of neurons in the third hidden layer = 5
  - Number of neurons in the output layer = 1
- Learning rate (α) = 0.0075
- Number of iterations = 2500

Graph between cost and number of iterations



With increase in the number of iterations our cost decreases and comes close to 0, i.e. our objective to minimize the cost function is achieved.

Training set accuracy achieved = 98.56%

Test set accuracy achieved = 80%

A few types 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)

9. Conclusion and future work

Our Neural Network model performs very well on the training dataset with an accuracy of 98.56% while it performs decently on the test dataset with an accuracy of 80%. However, our Neural Network model classifies some images incorrectly which may include improperly scaled images or unusual images. These incorrect classifications can be minimized by improving our Neural Network model performance by using various other cost optimization algorithms in place of gradient descent algorithm such as – gradient descent with momentum, RMSProp, Adam or Adaptive Moment Optimization algorithms.