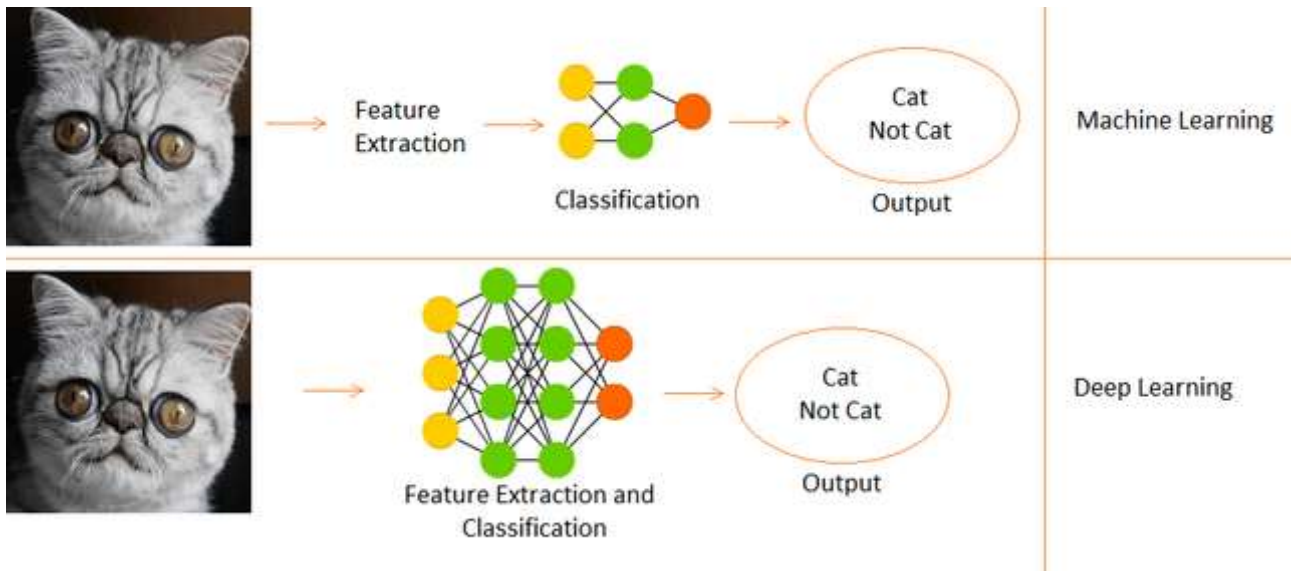
Deep learning tutorial- Logistic Regression and Artificial Neural Network

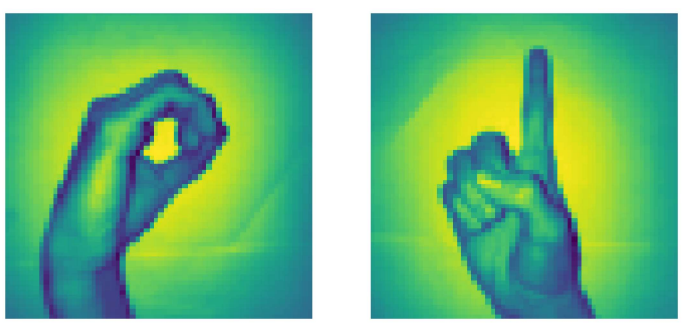
- **Why deep learning:** When the amounth of data is increased, machine learning techniques are insufficient in terms of performance and deep learning gives better performance like accuracy.
- **Usage fields of deep learning:** Speech recognition, image classification, natural language procession (nlp) or recommendation systems



Load datasets

In [2]:
```
x_l = np.load('../input/Sign-language-digits-dataset/X.npy')
Y_l = np.load('../input/Sign-language-digits-dataset/Y.npy')
img_size = 64
plt.subplot(1, 2, 1)
plt.imshow(x_l[260].reshape(img_size, img_size))
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(x_l[900].reshape(img_size, img_size))
plt.axis('off')
```

Out[2]:
```
(-0.5, 63.5, 63.5, -0.5)
```



- In order to create image array, I concatenate zero sign and one sign arrays
- Then I create label array 0 for zero sign images and 1 for one sign images.

In [3]:
```
# Join a sequence of arrays along an row axis.
X = np.concatenate((x_l[204:409], x_l[822:1027] ), axis=0) # from 0 to 204 is zero sign and from 205 to 410 is one sign
z = np.zeros(205)
o = np.ones(205)
Y = np.concatenate((z, o), axis=0).reshape(X.shape[0],1)
print("X shape: " , X.shape)
print("Y shape: " , Y.shape)
```

```
X shape:  (410, 64, 64)
Y shape:  (410, 1)
```

- The shape of the X is (410, 64, 64)
  - 410 means that we have 410 images (zero and one signs)
  - 64 means that our image size is 64x64 (64x64 pixels)
- The shape of the Y is (410,1)
  - 410 means that we have 410 labels (0 and 1)
- Lets split X and Y into train and test sets.
  - test_size = percentage of test size. test = 15% and train = 75%
  - random_state = use same seed while randomizing. It means that if we call train_test_split repeatedly, it always creates same train and test distribution because we have same random_state.

In [4]:
```
# Then lets create x_train, y_train, x_test, y_test arrays
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.15, random_state=42)
```

```
number_of_train = X_train.shape[0]
number_of_test = X_test.shape[0]
```

- Now we have 3 dimensional input array (X) so we need to make it flatten (2D) in order to use as input for our first deep learning model.
- Our label array (Y) is already flatten(2D) so we leave it like that.
- Lets flatten X array(images array).

In [5]:
```
X_train_flatten = X_train.reshape(number_of_train,X_train.shape[1]*X_train.shape[2])
X_test_flatten = X_test .reshape(number_of_test,X_test.shape[1]*X_test.shape[2])
print("X train flatten",X_train_flatten.shape)
print("X test flatten",X_test_flatten.shape)
```

```
X train flatten (348, 4096)
X test flatten (62, 4096)
```

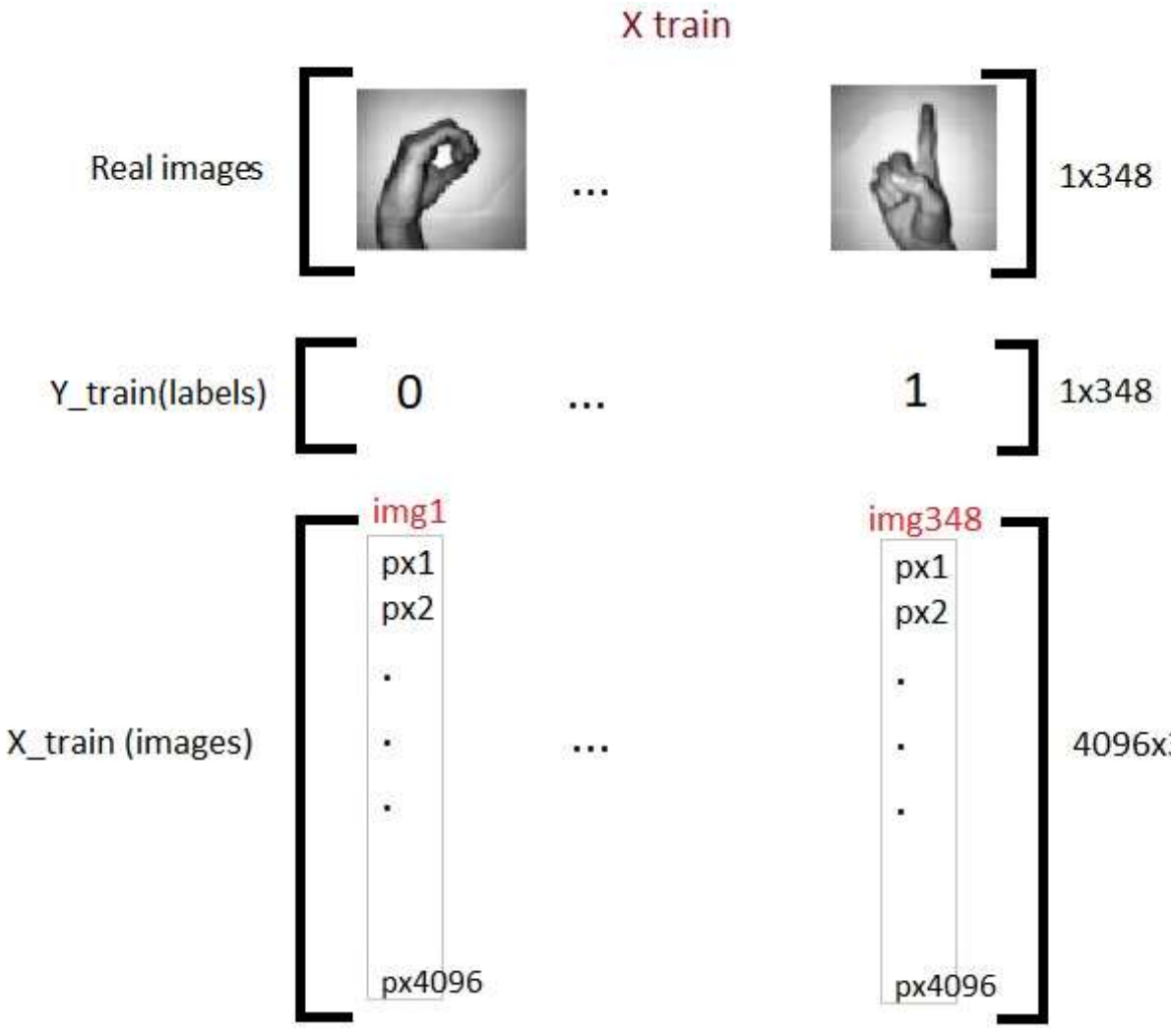
- As you can see, we have 348 images and each image has 4096 pixels in image train array.
- Also, we have 62 images and each image has 4096 pixels in image test array.
- Then lets take transpose.

In [6]:
```
x_train = X_train_flatten.T
x_test = X_test_flatten.T
y_train = Y_train.T
y_test = Y_test.T
print("x train: ",x_train.shape)
print("x test: ",x_test.shape)
print("y train: ",y_train.shape)
print("y test: ",y_test.shape)
```

```
x train:  (4096, 348)
x test:  (4096, 62)
y train:  (1, 348)
y test:  (1, 62)
```

What we did up to this point:

- Choose our labels (classes) that are sign zero and sign one
- Create and flatten train and test sets
- Our final inputs(images) and outputs(labels or classes) looks like this:



Logistic Regression

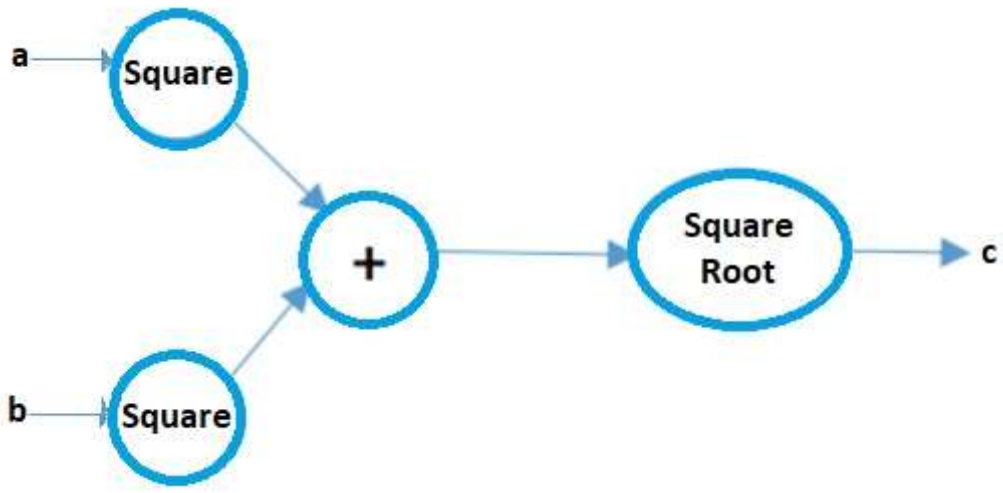
- When we talk about binary classification(0 and 1 outputs) what comes to mind first is logistic regression.
- logistic regression is actually a very simple neural network.
- By the way neural network and deep learning are same thing. When we will come artificial neural network.
- In order to understand logistic regression (simple deep learning) lets first learn computation graph.

Computation Graph

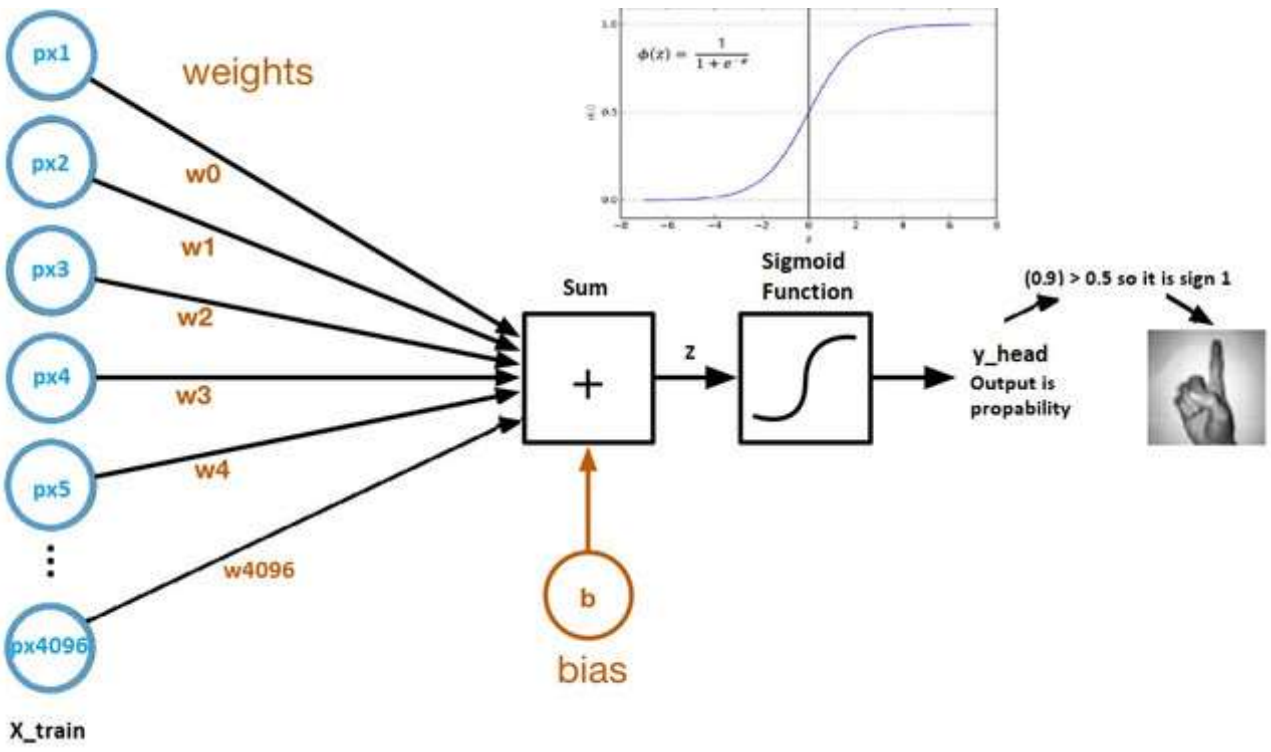
- Computation graphs are a nice way to think about mathematical expressions.
- It is like visualization of mathematical expressions.
- For example we have

$$c = \sqrt{a^2 + b^2}$$

- It's computational graph is this. As you can see we express math with graph.



Computation graph of logistic regression




```
* Parameters are weight and bias.
* Weights: coefficients of each pixels
* Bias: intercept
* z = (w.t)x + b  => z equals to (transpose of weights times input x) + bias
* In an other saying => z = b + px1*w1 + px2*w2 + ... + px4096*w4096
* y_head = sigmoid(z)
* Sigmoid function makes z between zero and one so that is probability. You can see sigmoid function
in computation graph.
```

- Why we use sigmoid function?
  - It gives probabilistic result
  - It is derivative so we can use it in gradient descent algorithm.
- Lets make example:
  - Lets say we find z = 4 and put z into sigmoid function. The result(y_head) is almost 0.9. It means that our classification result is 1 with 90% probability.
- Now lets start with from beginning and examine each component of computation graph more detailed.

Initializing parameters

- As you know input is our images that has 4096 pixels(each image in x_train).
- Each pixels have own weights.
- The first step is multiplying each pixels with their own weights.
- The question is that what is the initial value of weights?
  - There are some techniques that I will explain at artificial neural network but for this time initial weights are 0.01.
  - Okey, weights are 0.01 but what is the weight array shape? As you understand from computation graph of logistic regression, it is (4096,1)
  - Also initial bias is 0.
- Lets write some code. In order to use at coming topics like artificial neural network (ANN).

In [7]:
```
# short description and example of definition (def)
def dummy(parameter):
    dummy_parameter = parameter + 5
    return dummy_parameter
result = dummy(3)     # result = 8

# lets initialize parameters
# So what we need is dimension 4096 that is number of pixels as a parameter for our initialize method(def)
```

```
def initialize_weights_and_bias(dimension):
    w = np.full((dimension,1),0.01)
    b = 0.0
    return w, b
```

In [8]:
```
#w,b = initialize_weights_and_bias(4096)
```

Forward Propagation

- The all steps from pixels to cost is called forward propagation
  - z = (w.T)x + b => in this equation we know x that is pixel array, we know w (weights) and b (bias) so the rest is calculation. (T is transpose)
  - Then we put z into sigmoid function that returns y_head(probability). When your mind is confused go and look at computation graph. Also equation of sigmoid function is in computation graph.
  - Then we calculate loss(error) function.
  - Cost function is summation of all loss(error).
  - Lets start with z and the write sigmoid definition(method) that takes z as input parameter and returns y_head(probability)

In [9]:
```
# calculation of z
#z = np.dot(w.T,x_train)+b
def sigmoid(z):
    y_head = 1/(1+np.exp(-z))
    return y_head
```

In [10]:
```
y_head = sigmoid(0)
y_head
```

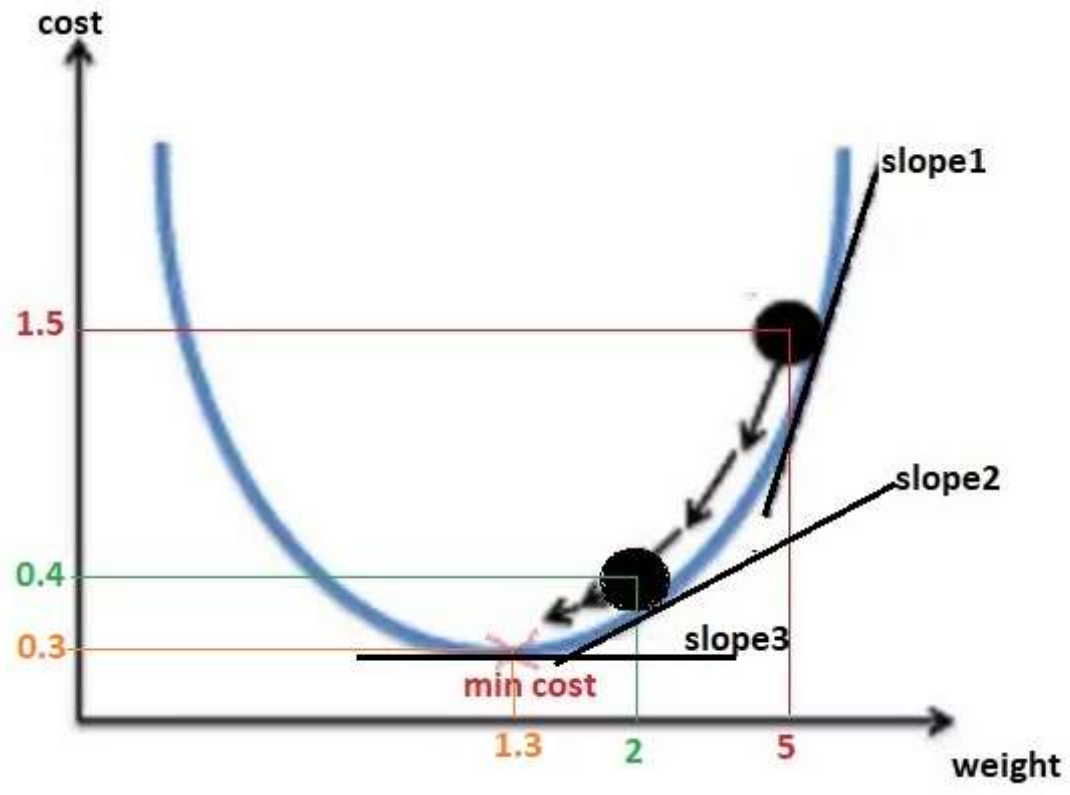
Out[10]:
```
0.5
```

- As we write sigmoid method and calculate y_head. Lets learn what is loss(error) function
- Lets make example, I put one image as input then multiply it with their weights and add bias term so I find z. Then put z into sigmoid method so I find y_head. Up to this point we know what we did. Then e.g y_head became 0.9 that is bigger than 0.5 so our prediction is image is sign one image. Okey every thing looks like fine. But, is our prediction is correct and how do we check whether it is correct or not? The answer is with loss(error) function:
  - Mathematical expression of log loss(error) function is that: $-(1-y)\log(1-\hat{y}) - y\log\hat{y}$
  - It says that if you make wrong prediction, loss(error) becomes big. **DENKLEM DUZELTME**
    - Example: our real image is sign one and its label is 1 (y = 1), then we make prediction y_head = 1. When we put y and y_head into loss(error) equation the result is 0. We make correct prediction therefore our loss is 0. However, if we make wrong prediction like y_head = 0, loss(error) is infinity.
- After that, the cost function is summation of loss function. Each image creates loss function. Cost function is summation of loss functions that is created by each input image.
- Lets implement forward propagation.

In [11]:
```
# Forward propagation steps:
# find z = w.T*x+b
# y_head = sigmoid(z)
# loss(error) = loss(y,y_head)
# cost = sum(loss)
def forward_propagation(w,b,x_train,y_train):
    z = np.dot(w.T,x_train) + b
    y_head = sigmoid(z) # probabilistic 0-1
    loss = -y_train*np.log(y_head)-(1-y_train)*np.log(1-y_head)
    cost = (np.sum(loss))/x_train.shape[1]      # x_train.shape[1]  is for scaling
    return cost
```

Optimization Algorithm with Gradient Descent

- Well, now we know what is our cost that is error.
- Therefore, we need to decrease cost because as we know if cost is high it means that we make wrong prediction.
- Lets think first step, every thing starts with initializing weights and bias. Therefore cost is dependent with them.
- In order to decrease cost, we need to update weights and bias.
- In other words, our model needs to learn the parameters weights and bias that minimize cost function. This technique is called gradient descent.
- Lets make an example:
  - We have w = 5 and bias = 0 (so ignore bias for now). Then we make forward propagation and our cost function is 1.5.



- It looks like this. (red lines)
- As you can see from graph, we are not at minimum point of cost function. Therefore we need to go through minimum cost. Okey, lets update weight. (the symbol := is updating)
- w := w - step. The question is what is this step? Step is slope1. Okey, it looks remarkable. In order to find minimum point, we can use slope1. Then lets say slope1 = 3 and update our weight. w := w - slope1 => w = 2.
- Now, our weight w is 2. As you remember, we need to find cost function with forward propagation again.
- Lets say according to forward propagation with w = 2, cost function is 0.4. Hmm, we are at right way because our cost function is decrease. We have new value for cost function that is cost = 0.4. Is that enough? Actually I do not know lets try one more step.
- Slope2 = 0.7 and w = 2. Lets update weight w : = w - step(slope2) => w = 1.3 that is new weight. So lets find new cost.
- Make one more forward propagation with w = 1.3 and our cost = 0.3. Okey, our cost even decreased, it looks like fine but is it enough or do we need to make one more step? The answer is again I do not know, lets try.
- Slope3 = 0.01 and w = 1.3. Updating weight w := w - step(slope3) => w = 1.29 ~ 1.3. So weight does not change because we find minimum point of cost function.
- Everything looks like good but how we find slope? If you remember from high school or university, in order to find slope of function(cost function) at given point(at given weight) we take derivative of function at given point. Also you can ask that okey well we find slope but how it knows where it go. You can say that it can go more higher cost values instead of going minimum point. The asnwer is that slope(derivative) gives both step and direction of step. Therefore do not worry :)
- Update equation is this. It says that there is a cost function(takes weight and bias). Take derivative of cost function according to weight and bias. Then multiply it with α learning rate. Then update weight. (In order to explain I ignore bias but these all

$$w := w - \alpha \frac{\partial J(w,b)}{\partial (w,b)}$$

  steps will be applied for bias)
- Now, I am sure you are asking what is learning rate that I mentioned never. It is very simple term that determines learning rate. Hovewer there is tradeoff between learning fast and never learning. For example you are at Paris(current cost) and want to go Madrid(minimum cost). If your speed(learning rate) is small, you can go Madrid very slowly and it takes too long time. On ther other hand, if your speed(learning rate) is big, you can go very fast but maybe you make crash and never go to Madrid. Therefore, we need to choose wisely our speed(learning rate).
- Learning rate is also called hyperparameter that need to be chosen and tuned. I will explain it more detailed in artificial neural network with other hyperparameters. For now just say learning rate is 1 for our previous example.
- I think now you understand the logic behind forward propagation(from weights and bias to cost) and backward propagation(from cost to weights and bias to update them). Also you learn gradient descent. Before implementing the code you need to learn one more thing that is how we take derivative of cost function according to weights and bias. It is not related with python or coding. It is pure mathematic. There are two option first one is to google how to take derivative of log loss function and second one is even to google what is derivative of log loss function :) I choose second one because I cannot explain math without talking :)

$$\frac{\partial J}{\partial w} = \frac{1}{m} x(y_head - y)^T$$

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

In [12]:
```
# In backward propagation we will use y_head that found in forward progation
# Therefore instead of writing backward propagation method, lets combine forward propagation and backward propagation
def forward_backward_propagation(w,b,x_train,y_train):
    # forward propagation
    z = np.dot(w.T,x_train) + b
    y_head = sigmoid(z)
    loss = -y_train*np.log(y_head)-(1-y_train)*np.log(1-y_head)
    cost = (np.sum(loss))/x_train.shape[1]      # x_train.shape[1]  is for scaling
    # backward propagation
    derivative_weight = (np.dot(x_train,((y_head-y_train).T)))/x_train.shape[1] # x_train.shape[1]  is for scaling
    derivative_bias = np.sum(y_head-y_train)/x_train.shape[1]                 # x_train.shape[1]  is for scaling
    gradients = {"derivative_weight": derivative_weight,"derivative_bias": derivative_bias}
    return cost,gradients
```

- Up to this point we learn
  - Initializing parameters (implemented)
  - Finding cost with forward propagation and cost function (implemented)
  - Updating(learning) parameters (weight and bias). Now lets implement it.

In [13]:
```
# Updating(learning) parameters
def update(w, b, x_train, y_train, learning_rate,number_of_iterarion):
    cost_list = []
    cost_list2 = []
    index = []
    # updating(learning) parameters is number_of_iterarion times
    for i in range(number_of_iterarion):
        # make forward and backward propagation and find cost and gradients
        cost,gradients = forward_backward_propagation(w,b,x_train,y_train)
        cost_list.append(cost)
        # lets update
        w = w - learning_rate * gradients["derivative_weight"]
        b = b - learning_rate * gradients["derivative_bias"]
        if i % 10 == 0:
            cost_list2.append(cost)
            index.append(i)
            print ("Cost after iteration %i: %f" %(i, cost))
    # we update(learn) parameters weights and bias
    parameters = {"weight": w,"bias": b}
    plt.plot(index,cost_list2)
    plt.xticks(index,rotation='vertical')
    plt.xlabel("Number of Iterarion")
    plt.ylabel("Cost")
    plt.show()
    return parameters, gradients, cost_list
#parameters, gradients, cost_list = update(w, b, x_train, y_train, learning_rate = 0.009,number_of_iterarion = 200)
```

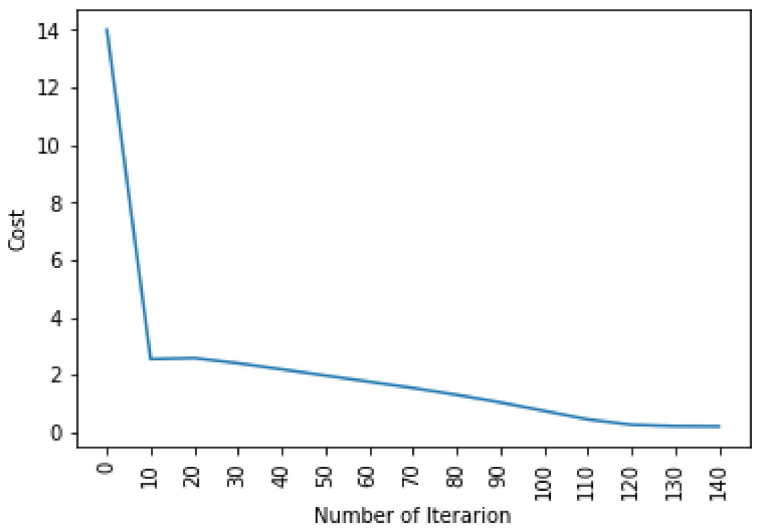
- Up to this point we learn our parameters. It means we fit the data.
- In order to predict we have parameters. Therefore, lets predict.
- In prediction step we have x_test as a input and while using it, we make forward prediction.

In [14]:
```
 # prediction
def predict(w,b,x_test):
    # x_test is a input for forward propagation
    z = sigmoid(np.dot(w.T,x_test)+b)
    Y_prediction = np.zeros((1,x_test.shape[1]))
    # if z is bigger than 0.5, our prediction is sign one (y_head=1),
    # if z is smaller than 0.5, our prediction is sign zero (y_head=0),
    for i in range(z.shape[1]):
        if z[0,i]<= 0.5:
            Y_prediction[0,i] = 0
        else:
            Y_prediction[0,i] = 1

    return Y_prediction
# predict(parameters["weight"],parameters["bias"],x_test)
```

- We make prediction.
- Now lets put them all together.

In [15]:
```
def logistic_regression(x_train, y_train, x_test, y_test, learning_rate ,  num_iterations):
    # initialize
    dimension =  x_train.shape[0]  # that is 4096
    w,b = initialize_weights_and_bias(dimension)
    # do not change learning rate
    parameters, gradients, cost_list = update(w, b, x_train, y_train, learning_rate,num_iterations)

    y_prediction_test = predict(parameters["weight"],parameters["bias"],x_test)
    y_prediction_train = predict(parameters["weight"],parameters["bias"],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))

logistic_regression(x_train, y_train, x_test, y_test,learning_rate = 0.01, num_iterations = 150)
```

```
Cost after iteration 0: 14.014222
Cost after iteration 10: 2.544689
Cost after iteration 20: 2.577950
Cost after iteration 30: 2.397999
Cost after iteration 40: 2.185019
Cost after iteration 50: 1.968348
Cost after iteration 60: 1.754195
Cost after iteration 70: 1.535079
Cost after iteration 80: 1.297567
Cost after iteration 90: 1.031919
Cost after iteration 100: 0.737019
Cost after iteration 110: 0.441355
Cost after iteration 120: 0.252278
Cost after iteration 130: 0.205168
Cost after iteration 140: 0.196168
```




```
train accuracy: 92.816091954023 %
test accuracy: 93.54838709677419 %
```

- We learn logic behind simple neural network(logistic regression) and how to implement it.
- Now that we have learned logic, we can use sklearn library which is easier than implementing all steps with hand for logistic regression.

Logistic Regression with Sklearn

- In sklearn library, there is a logistic regression method that ease implementing logistic regression.
- The accuracies are different from what we find. Because logistic regression method use a lot of different feature that we do not use like different optimization parameters or regularization.
- Lets make conclusion for logistic regression and continue with artificial neural network.

In [16]:
```
from sklearn import linear_model
logreg = linear_model.LogisticRegression(random_state = 42,max_iter= 150)
print("test accuracy: {} ".format(logreg.fit(x_train.T, y_train.T).score(x_test.T, y_test.T)))
print("train accuracy: {} ".format(logreg.fit(x_train.T, y_train.T).score(x_train.T, y_train.T)))
```

```
test accuracy: 0.967741935483871
train accuracy: 1.0
```

Summary

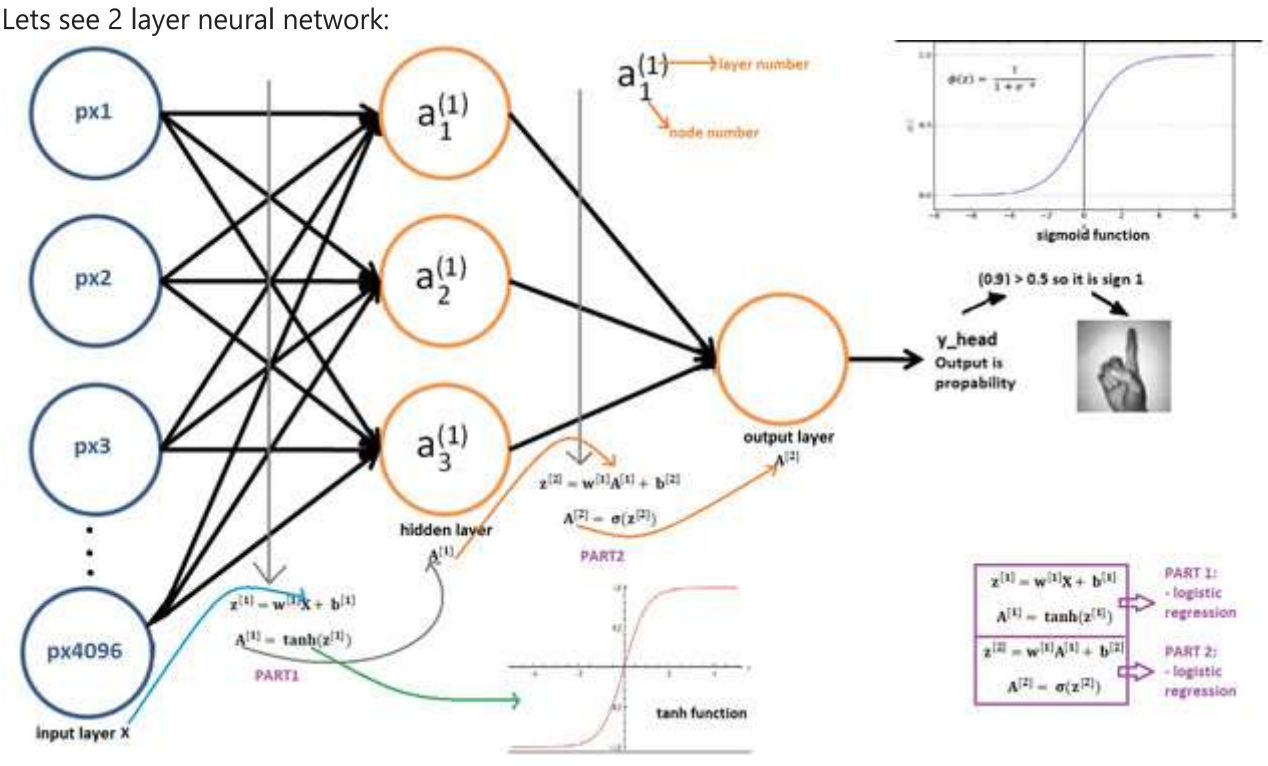
What we did at this first part:

- Initialize parameters weight and bias
- Forward propagation
- Loss function
- Cost function
- Backward propagation (gradient descent)
- Prediction with learnt parameters weight and bias
- Logistic regression with sklearn

Artificial Neural Network (ANN)

- It is also called deep neural network or deep learning.
- **What is neural network:** It is basically taking logistic regression and repeating it at least 2 times.
- In logistic regression, there are input and output layers. However, in neural network, there is at least one hidden layer between input and output layer.
- **What is deep, in order to say "deep" how many layer do I need to have:** "Deep" is a relative term; it refers to the "depth" of a network, meaning how many hidden layers it has. "How deep is your swimming pool?" could be 12 feet or it might be two feet; nevertheless, it still has a depth--it has the quality of "deepness". 32 years ago, I used two or three hidden layers. That was the limit for the specialized hardware of the day. Just a few years ago, 20 layers was considered pretty deep.
- **Why it is called hidden:** Because hidden layer does not see inputs(training set)
- For example you have input, one hidden and output layers. When someone ask you "hey my friend how many layers do your neural network have?" The answer is "I have 2 layer neural network". Because while computing layer number input layer is ignored.

- Lets see 2 layer neural network:



- Step by step we will learn this image.
  - As you can see there is one hidden layer between input and output layers. And this hidden layer has 3 nodes. If you are curious why I choose number of node 3, the answer is there is no reason, I only choose :). Number of node is hyperparameter like learning rate. Therefore we will see hyperparameters at the end of artificial neural network.
  - Input and output layers do not change. They are same like logistic regression.
  - In image, there is a tanh function that is unknown for you. It is a activation function like sigmoid function. Tanh activation function is better than sigmoid for hidden units bacause mean of its output is closer to zero so it centers the data better for the next layer. Also tanh activation function increase non linearity that cause our model learning better.
  - As you can see with purple color there are two parts. Both parts are like logistic regression. The only difference is activation function, inputs and outputs.
    - In logistic regression: input => output
    - In 2 layer neural network: input => hidden layer => output. You can think that hidden layer is output of part 1 and input of part 2.
- We will follow the same path like logistic regression for 2 layer neural network.

2-Layer Neural Network

- Size of layers and initializing parameters weights and bias
- Forward propagation
- Loss function and Cost function
- Backward propagation
- Update Parameters
- Prediction with learnt parameters weight and bias
- Create Model

Size of layers and initializing parameters weights and bias

- For x_train that has 348 sample $x^{(348)}$:

$$z^{[1](348)} = W^{[1]}x^{(348)} + b^{[1](348)}$$

$$a^{[1](348)} = \tanh(z^{[1](348)})$$

$$z^{[2](348)} = W^{[2]}a^{[1](348)} + b^{[2](348)}$$

$$\hat{y}^{(348)} = a^{[2](348)} = \sigma(z^{[2](348)})$$

- At logistic regression, we initialize weights 0.01 and bias 0. At this time, we initialize weights randomly. Because if we initialize parameters zero each neuron in the first hidden layer will perform the same comptation. Therefore, even after multiple iterartion of gradiet descent each neuron in the layer will be computing same things as other neurons. Therefore we initialize randomly. Also initial weights will be small. If they are very large initially, this will cause the inputs of the tanh to be very large, thus causing gradients to be close to zero. The optimization algorithm will be slow.

- Bias can be zero initially.

In [17]:
```
# intialize parameters and layer sizes
def initialize_parameters_and_layer_sizes_NN(x_train, y_train):
    parameters = {"weight1": np.random.randn(3,x_train.shape[0]) * 0.1,
                  "bias1": np.zeros((3,1)),
                  "weight2": np.random.randn(y_train.shape[0],3) * 0.1,
                  "bias2": np.zeros((y_train.shape[0],1))}
    return parameters
```

Forward propagation

- Forward propagation is almost same with logistic regression.
- The only difference is we use tanh function and we make all process twice.

- Also numpy has tanh function. So we do not need to implement it.

In [18]:
```
def forward_propagation_NN(x_train, parameters):

    Z1 = np.dot(parameters["weight1"],x_train) +parameters["bias1"]
    A1 = np.tanh(Z1)
    Z2 = np.dot(parameters["weight2"],A1) + parameters["bias2"]
    A2 = sigmoid(Z2)

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

    return A2, cache
```

Loss function and Cost function

- Loss and cost functions are same with logistic regression

$$J(\theta) = -\sum_i y_i ln(\hat{y}_i)$$

- Cross entropy function

In [19]:
```
# Compute cost
def compute_cost_NN(A2, Y, parameters):
    logprobs = np.multiply(np.log(A2),Y)
    cost = -np.sum(logprobs)/Y.shape[1]
    return cost
```

Backward propagation

- As you know backward propagation means derivative.

In [20]:
```
# Backward Propagation
def backward_propagation_NN(parameters, cache, X, Y):

    dZ2 = cache["A2"]-Y
    dW2 = np.dot(dZ2,cache["A1"].T)/X.shape[1]
    db2 = np.sum(dZ2,axis =1,keepdims=True)/X.shape[1]
    dZ1 = np.dot(parameters["weight2"].T,dZ2)*(1 - np.power(cache["A1"], 2))
    dW1 = np.dot(dZ1,X.T)/X.shape[1]
    db1 = np.sum(dZ1,axis =1,keepdims=True)/X.shape[1]
    grads = {"dweight1": dW1,
             "dbias1": db1,
             "dweight2": dW2,
             "dbias2": db2}
    return grads
```

Update Parameters

- Updating parameters also same with logistic regression.

In [21]:
```
# update parameters
def update_parameters_NN(parameters, grads, learning_rate = 0.01):
    parameters = {"weight1": parameters["weight1"]-learning_rate*grads["dweight1"],
                  "bias1": parameters["bias1"]-learning_rate*grads["dbias1"],
                  "weight2": parameters["weight2"]-learning_rate*grads["dweight2"],
                  "bias2": parameters["bias2"]-learning_rate*grads["dbias2"]}

    return parameters
```

Prediction with learnt parameters weight and bias

- Lets write predict method that is like logistic regression.

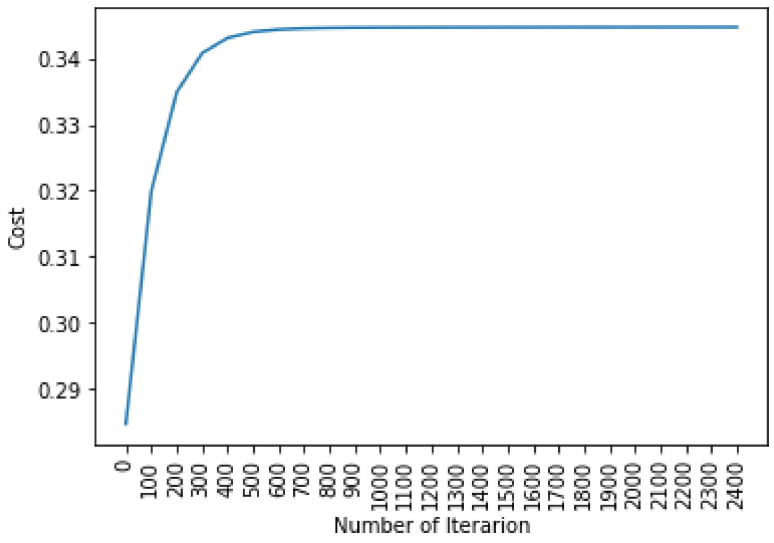
In [22]:
```
# prediction
def predict_NN(parameters,x_test):
    # x_test is a input for forward propagation
    A2, cache = forward_propagation_NN(x_test,parameters)
    Y_prediction = np.zeros((1,x_test.shape[1]))
    # if z is bigger than 0.5, our prediction is sign one (y_head=1),
    # if z is smaller than 0.5, our prediction is sign zero (y_head=0),
    for i in range(A2.shape[1]):
        if A2[0,i]<= 0.5:
            Y_prediction[0,i] = 0
        else:
            Y_prediction[0,i] = 1

    return Y_prediction
```

Create Model

- Lets put them all together

In [23]:
```
# 2 - Layer neural network
def two_layer_neural_network(x_train, y_train,x_test,y_test, num_iterations):
    cost_list = []
    index_list = []
    #initialize parameters and layer sizes
    parameters = initialize_parameters_and_layer_sizes_NN(x_train, y_train)

    for i in range(0, num_iterations):
         # forward propagation
        A2, cache = forward_propagation_NN(x_train,parameters)
        # compute cost
        cost = compute_cost_NN(A2, y_train, parameters)
         # backward propagation
        grads = backward_propagation_NN(parameters, cache, x_train, y_train)
         # update parameters
        parameters = update_parameters_NN(parameters, grads)

        if i % 100 == 0:
            cost_list.append(cost)
            index_list.append(i)
            print ("Cost after iteration %i: %f" %(i, cost))
    plt.plot(index_list,cost_list)
    plt.xticks(index_list,rotation='vertical')
    plt.xlabel("Number of Iterarion")
    plt.ylabel("Cost")
    plt.show()

    # predict
    y_prediction_test = predict_NN(parameters,x_test)
    y_prediction_train = predict_NN(parameters,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))
    return parameters

parameters = two_layer_neural_network(x_train, y_train,x_test,y_test, num_iterations=2500)
```

```
Cost after iteration 0: 0.284707
Cost after iteration 100: 0.319944
Cost after iteration 200: 0.334917
Cost after iteration 300: 0.340822
Cost after iteration 400: 0.343125
Cost after iteration 500: 0.344033
Cost after iteration 600: 0.344401
Cost after iteration 700: 0.344558
Cost after iteration 800: 0.344630
Cost after iteration 900: 0.344667
Cost after iteration 1000: 0.344689
Cost after iteration 1100: 0.344702
Cost after iteration 1200: 0.344712
Cost after iteration 1300: 0.344719
Cost after iteration 1400: 0.344725
Cost after iteration 1500: 0.344730
Cost after iteration 1600: 0.344734
Cost after iteration 1700: 0.344737
Cost after iteration 1800: 0.344740
Cost after iteration 1900: 0.344742
Cost after iteration 2000: 0.344745
Cost after iteration 2100: 0.344746
Cost after iteration 2200: 0.344748
Cost after iteration 2300: 0.344750
Cost after iteration 2400: 0.344751
```




```
train accuracy: 50.57471264367816 %
test accuracy: 46.7741935483871 %
```

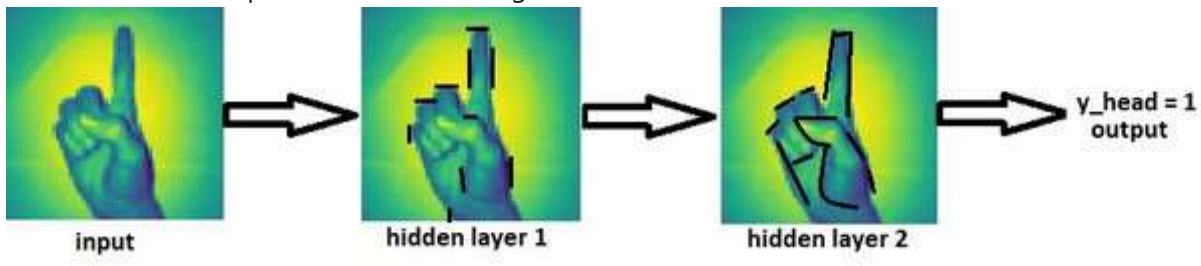
Up to this point we create 2 layer neural network and learn how to implement

- Size of layers and initializing parameters weights and bias
- Forward propagation
- Loss function and Cost function
- Backward propagation
- Update Parameters
- Prediction with learnt parameters weight and bias
- Create Model

Now lets learn how to implement L layer neural network with keras.

L Layer Neural Network

- **What happens if number of hidden layer increase:** Earlier layers can detect simple features.
- When model composing simple features together in later layers of neural network that it can learn more and more complex functions. For example, lets look at our sign one.



- For example first hidden layer learns edges or basic shapes like line. When number of layer increase, layers start to learn more complex things like convex shapes or characteristic features like forefinger.
- Lets create our model
  - There are some hyperparameters we need to choose like learning rate, number of iterations, number of hidden layer, number of hidden units, type of activation functions. Woww it is too much :)
  - These hyperparameters can be chosen intiutively if you spend a lot of time in deep learning world.
  - However, if you do not spend too much time, the best way is to google it but it is not necessary. You need to try hyperparameters to find best one.
  - In this tutorial our model will have 2 hidden layer with 8 and4 nodes, respectively. Because when number of hidden layer and node increase, it takes too much time.
  - As a activation function we will use relu(first hidden layer), relu(second hidden layer) and sigmoid(output layer) respectively.
  - Number of iteration will be 100.
- Our way is same with previous parts however as you learn the logic behind deep learning, we can ease our job and use keras library for deeper neural networks.
- First lets reshape our x_train, x_test, y_train and y_test.

In [24]:
```
# reshaping
x_train, x_test, y_train, y_test = x_train.T, x_test.T, y_train.T, y_test.T
```

Implementing with keras library

Lets look at some parameters of keras library:

- units: output dimensions of node
- kernel_initializer: to initialize weights
- activation: activation function, we use relu
- input_dim: input dimension that is number of pixels in our images (4096 px)
- optimizer: we use adam optimizer
  - Adam is one of the most effective optimization algorithms for training neural networks.
  - Some advantages of Adam is that relatively low memory requirements and usually works well even with little tuning of hyperparameters
- loss: Cost function is same. By the way the name of the cost function is cross-entropy cost function that we use previous parts.

$$J = -\frac{1}{m}\sum_{i=0}^{m}\left(y^{(i)}\log\left(a^{[2](i)}\right) + (1-y^{(i)})\log\left(1-a^{[2](i)}\right)\right) \tag{6}$$

- metrics: it is accuracy.
- cross_val_score: use cross validation.
- epochs: number of iteration

In [25]:
```
# Evaluating the ANN
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
from keras.models import Sequential # initialize neural network library
from keras.layers import Dense # build our layers library
def build_classifier():
    classifier = Sequential() # initialize neural network
    classifier.add(Dense(units = 8, kernel_initializer = 'uniform', activation = 'relu', input_dim = x_train.shape[1]))
    classifier.add(Dense(units = 4, kernel_initializer = 'uniform', activation = 'relu'))
    classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activation = 'sigmoid'))
    classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
    return classifier
classifier = KerasClassifier(build_fn = build_classifier, epochs = 100)
accuracies = cross_val_score(estimator = classifier, X = x_train, y = y_train, cv = 3)
mean = accuracies.mean()
variance = accuracies.std()
print("Accuracy mean: "+ str(mean))
print("Accuracy variance: "+ str(variance))
```

```
Using TensorFlow backend.
```

```
WARNING:tensorflow:From /opt/conda/lib/python3.6/site-packages/tensorflow/python/framework/op_def_library.py:263: coloc
ate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
WARNING:tensorflow:From /opt/conda/lib/python3.6/site-packages/tensorflow/python/ops/math_ops.py:3066: to_int32 (from t
ensorflow.python.ops.math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
Epoch 1/100
232/232 [==============================] - 3s 12ms/step - loss: 0.6919 - acc: 0.5431
Epoch 2/100
232/232 [==============================] - 0s 95us/step - loss: 0.6892 - acc: 0.5431
Epoch 3/100
232/232 [==============================] - 0s 100us/step - loss: 0.6866 - acc: 0.5431
Epoch 4/100
232/232 [==============================] - 0s 87us/step - loss: 0.6846 - acc: 0.5431
Epoch 5/100
232/232 [==============================] - 0s 97us/step - loss: 0.6813 - acc: 0.5431
Epoch 6/100
232/232 [==============================] - 0s 97us/step - loss: 0.6754 - acc: 0.5431
Epoch 7/100
232/232 [==============================] - 0s 95us/step - loss: 0.6685 - acc: 0.5905
Epoch 8/100
232/232 [==============================] - 0s 99us/step - loss: 0.6573 - acc: 0.6681
Epoch 9/100
232/232 [==============================] - 0s 88us/step - loss: 0.6468 - acc: 0.5474
Epoch 10/100
232/232 [==============================] - 0s 91us/step - loss: 0.6271 - acc: 0.8233
Epoch 11/100
232/232 [==============================] - 0s 88us/step - loss: 0.6082 - acc: 0.6379
Epoch 12/100
232/232 [==============================] - 0s 92us/step - loss: 0.5683 - acc: 0.8362
Epoch 13/100
232/232 [==============================] - 0s 97us/step - loss: 0.5400 - acc: 0.7974
Epoch 14/100
232/232 [==============================] - 0s 92us/step - loss: 0.5006 - acc: 0.9009
Epoch 15/100
232/232 [==============================] - 0s 88us/step - loss: 0.4623 - acc: 0.8405
Epoch 16/100
232/232 [==============================] - 0s 101us/step - loss: 0.4204 - acc: 0.9095
Epoch 17/100
232/232 [==============================] - 0s 107us/step - loss: 0.3837 - acc: 0.9052
Epoch 18/100
232/232 [==============================] - 0s 107us/step - loss: 0.3446 - acc: 0.9009
Epoch 19/100
232/232 [==============================] - 0s 105us/step - loss: 0.3129 - acc: 0.9267
Epoch 20/100
232/232 [==============================] - 0s 103us/step - loss: 0.2908 - acc: 0.9224
Epoch 21/100
232/232 [==============================] - 0s 102us/step - loss: 0.2789 - acc: 0.8966
Epoch 22/100
232/232 [==============================] - 0s 104us/step - loss: 0.2726 - acc: 0.9052
Epoch 23/100
232/232 [==============================] - 0s 104us/step - loss: 0.2285 - acc: 0.9310
Epoch 24/100
232/232 [==============================] - 0s 97us/step - loss: 0.2179 - acc: 0.9181
Epoch 25/100
232/232 [==============================] - 0s 98us/step - loss: 0.2121 - acc: 0.9267
Epoch 26/100
232/232 [==============================] - 0s 98us/step - loss: 0.2060 - acc: 0.9267
Epoch 27/100
232/232 [==============================] - 0s 93us/step - loss: 0.1879 - acc: 0.9310
Epoch 28/100
232/232 [==============================] - 0s 90us/step - loss: 0.1883 - acc: 0.9095
Epoch 29/100
232/232 [==============================] - 0s 96us/step - loss: 0.1777 - acc: 0.9397
Epoch 30/100
232/232 [==============================] - 0s 89us/step - loss: 0.1709 - acc: 0.9310
Epoch 31/100
232/232 [==============================] - 0s 89us/step - loss: 0.1850 - acc: 0.9181
Epoch 32/100
232/232 [==============================] - 0s 93us/step - loss: 0.1855 - acc: 0.9353
Epoch 33/100
232/232 [==============================] - 0s 97us/step - loss: 0.2160 - acc: 0.9224
Epoch 34/100
232/232 [==============================] - 0s 99us/step - loss: 0.1672 - acc: 0.9397
Epoch 35/100
232/232 [==============================] - 0s 102us/step - loss: 0.1546 - acc: 0.9353
Epoch 36/100
232/232 [==============================] - 0s 88us/step - loss: 0.1491 - acc: 0.9526
Epoch 37/100
232/232 [==============================] - 0s 88us/step - loss: 0.1350 - acc: 0.9397
Epoch 38/100
232/232 [==============================] - 0s 89us/step - loss: 0.1410 - acc: 0.9569
Epoch 39/100
232/232 [==============================] - 0s 91us/step - loss: 0.1403 - acc: 0.9483
Epoch 40/100
232/232 [==============================] - 0s 98us/step - loss: 0.1444 - acc: 0.9483
Epoch 41/100
232/232 [==============================] - 0s 88us/step - loss: 0.1222 - acc: 0.9655
Epoch 42/100
232/232 [==============================] - 0s 91us/step - loss: 0.1266 - acc: 0.9483
Epoch 43/100
232/232 [==============================] - 0s 94us/step - loss: 0.1176 - acc: 0.9698
Epoch 44/100
```

```
232/232 [==============================] - 0s 95us/step - loss: 0.1307 - acc: 0.9483
Epoch 45/100
232/232 [==============================] - 0s 97us/step - loss: 0.1156 - acc: 0.9569
Epoch 46/100
232/232 [==============================] - 0s 93us/step - loss: 0.1324 - acc: 0.9526
Epoch 47/100
232/232 [==============================] - 0s 93us/step - loss: 0.1139 - acc: 0.9440
Epoch 48/100
232/232 [==============================] - 0s 94us/step - loss: 0.1111 - acc: 0.9655
Epoch 49/100
232/232 [==============================] - 0s 97us/step - loss: 0.1052 - acc: 0.9612
Epoch 50/100
232/232 [==============================] - 0s 95us/step - loss: 0.1053 - acc: 0.9612
Epoch 51/100
232/232 [==============================] - 0s 92us/step - loss: 0.1247 - acc: 0.9526
Epoch 52/100
232/232 [==============================] - 0s 91us/step - loss: 0.1132 - acc: 0.9569
Epoch 53/100
232/232 [==============================] - 0s 95us/step - loss: 0.1110 - acc: 0.9655
Epoch 54/100
232/232 [==============================] - 0s 95us/step - loss: 0.1187 - acc: 0.9612
Epoch 55/100
232/232 [==============================] - 0s 93us/step - loss: 0.1099 - acc: 0.9526
Epoch 56/100
232/232 [==============================] - 0s 87us/step - loss: 0.0993 - acc: 0.9698
Epoch 57/100
232/232 [==============================] - 0s 92us/step - loss: 0.0992 - acc: 0.9612
Epoch 58/100
232/232 [==============================] - 0s 89us/step - loss: 0.0936 - acc: 0.9569
Epoch 59/100
232/232 [==============================] - 0s 96us/step - loss: 0.0930 - acc: 0.9698
Epoch 60/100
232/232 [==============================] - 0s 99us/step - loss: 0.0861 - acc: 0.9655
Epoch 61/100
232/232 [==============================] - 0s 102us/step - loss: 0.0831 - acc: 0.9698
Epoch 62/100
232/232 [==============================] - 0s 94us/step - loss: 0.0831 - acc: 0.9655
Epoch 63/100
232/232 [==============================] - 0s 92us/step - loss: 0.0984 - acc: 0.9612
Epoch 64/100
232/232 [==============================] - 0s 93us/step - loss: 0.0904 - acc: 0.9741
Epoch 65/100
232/232 [==============================] - 0s 92us/step - loss: 0.1084 - acc: 0.9483
Epoch 66/100
232/232 [==============================] - 0s 93us/step - loss: 0.1487 - acc: 0.9526
Epoch 67/100
232/232 [==============================] - 0s 93us/step - loss: 0.1432 - acc: 0.9353
Epoch 68/100
232/232 [==============================] - 0s 90us/step - loss: 0.1380 - acc: 0.9483
Epoch 69/100
232/232 [==============================] - 0s 99us/step - loss: 0.0994 - acc: 0.9698
Epoch 70/100
232/232 [==============================] - 0s 99us/step - loss: 0.0818 - acc: 0.9698
Epoch 71/100
232/232 [==============================] - 0s 97us/step - loss: 0.0698 - acc: 0.9655
Epoch 72/100
232/232 [==============================] - 0s 101us/step - loss: 0.0741 - acc: 0.9741
Epoch 73/100
232/232 [==============================] - 0s 97us/step - loss: 0.0771 - acc: 0.9741
Epoch 74/100
232/232 [==============================] - 0s 99us/step - loss: 0.0744 - acc: 0.9698
Epoch 75/100
232/232 [==============================] - 0s 94us/step - loss: 0.0705 - acc: 0.9698
Epoch 76/100
232/232 [==============================] - 0s 93us/step - loss: 0.0693 - acc: 0.9741
Epoch 77/100
232/232 [==============================] - 0s 93us/step - loss: 0.0697 - acc: 0.9698
Epoch 78/100
232/232 [==============================] - 0s 90us/step - loss: 0.0715 - acc: 0.9828
Epoch 79/100
232/232 [==============================] - 0s 90us/step - loss: 0.0689 - acc: 0.9784
Epoch 80/100
232/232 [==============================] - 0s 90us/step - loss: 0.0654 - acc: 0.9698
Epoch 81/100
232/232 [==============================] - 0s 97us/step - loss: 0.0623 - acc: 0.9741
Epoch 82/100
232/232 [==============================] - 0s 90us/step - loss: 0.0654 - acc: 0.9698
Epoch 83/100
232/232 [==============================] - 0s 93us/step - loss: 0.0617 - acc: 0.9741
Epoch 84/100
232/232 [==============================] - 0s 92us/step - loss: 0.0601 - acc: 0.9828
Epoch 85/100
232/232 [==============================] - 0s 90us/step - loss: 0.0582 - acc: 0.9741
Epoch 86/100
232/232 [==============================] - 0s 89us/step - loss: 0.0657 - acc: 0.9828
Epoch 87/100
232/232 [==============================] - 0s 89us/step - loss: 0.0600 - acc: 0.9784
Epoch 88/100
232/232 [==============================] - 0s 93us/step - loss: 0.0652 - acc: 0.9784
Epoch 89/100
232/232 [==============================] - 0s 89us/step - loss: 0.0633 - acc: 0.9741
Epoch 90/100
232/232 [==============================] - 0s 90us/step - loss: 0.0569 - acc: 0.9784
Epoch 91/100
232/232 [==============================] - 0s 96us/step - loss: 0.0547 - acc: 0.9784
```

```
Epoch 92/100
232/232 [==============================] - 0s 92us/step - loss: 0.0491 - acc: 0.9871
Epoch 93/100
232/232 [==============================] - 0s 92us/step - loss: 0.0475 - acc: 0.9828
Epoch 94/100
232/232 [==============================] - 0s 92us/step - loss: 0.0544 - acc: 0.9741
Epoch 95/100
232/232 [==============================] - 0s 90us/step - loss: 0.0480 - acc: 0.9784
Epoch 96/100
232/232 [==============================] - 0s 97us/step - loss: 0.0529 - acc: 0.9741
Epoch 97/100
232/232 [==============================] - 0s 99us/step - loss: 0.0723 - acc: 0.9741
Epoch 98/100
232/232 [==============================] - 0s 92us/step - loss: 0.0978 - acc: 0.9655
Epoch 99/100
232/232 [==============================] - 0s 90us/step - loss: 0.0549 - acc: 0.9828
Epoch 100/100
232/232 [==============================] - 0s 101us/step - loss: 0.0506 - acc: 0.9828
116/116 [==============================] - 0s 364us/step
Epoch 1/100
232/232 [==============================] - 0s 1ms/step - loss: 0.6933 - acc: 0.4612
Epoch 2/100
232/232 [==============================] - 0s 90us/step - loss: 0.6931 - acc: 0.5043
Epoch 3/100
232/232 [==============================] - 0s 94us/step - loss: 0.6931 - acc: 0.5216
Epoch 4/100
232/232 [==============================] - 0s 89us/step - loss: 0.6931 - acc: 0.5216
Epoch 5/100
232/232 [==============================] - 0s 94us/step - loss: 0.6930 - acc: 0.5216
Epoch 6/100
232/232 [==============================] - 0s 99us/step - loss: 0.6930 - acc: 0.5216
Epoch 7/100
232/232 [==============================] - 0s 87us/step - loss: 0.6930 - acc: 0.5216
Epoch 8/100
232/232 [==============================] - 0s 93us/step - loss: 0.6929 - acc: 0.5216
Epoch 9/100
232/232 [==============================] - 0s 94us/step - loss: 0.6929 - acc: 0.5216
Epoch 10/100
232/232 [==============================] - 0s 94us/step - loss: 0.6929 - acc: 0.5216
Epoch 11/100
232/232 [==============================] - 0s 94us/step - loss: 0.6928 - acc: 0.5216
Epoch 12/100
232/232 [==============================] - 0s 95us/step - loss: 0.6928 - acc: 0.5216
Epoch 13/100
232/232 [==============================] - 0s 95us/step - loss: 0.6928 - acc: 0.5216
Epoch 14/100
232/232 [==============================] - 0s 102us/step - loss: 0.6928 - acc: 0.5216
Epoch 15/100
232/232 [==============================] - 0s 95us/step - loss: 0.6928 - acc: 0.5216
Epoch 16/100
232/232 [==============================] - 0s 98us/step - loss: 0.6927 - acc: 0.5216
Epoch 17/100
232/232 [==============================] - 0s 96us/step - loss: 0.6927 - acc: 0.5216
Epoch 18/100
232/232 [==============================] - 0s 92us/step - loss: 0.6927 - acc: 0.5216
Epoch 19/100
232/232 [==============================] - 0s 89us/step - loss: 0.6927 - acc: 0.5216
Epoch 20/100
232/232 [==============================] - 0s 88us/step - loss: 0.6926 - acc: 0.5216
Epoch 21/100
232/232 [==============================] - 0s 94us/step - loss: 0.6926 - acc: 0.5216
Epoch 22/100
232/232 [==============================] - 0s 94us/step - loss: 0.6927 - acc: 0.5216
Epoch 23/100
232/232 [==============================] - 0s 105us/step - loss: 0.6926 - acc: 0.5216
Epoch 24/100
232/232 [==============================] - 0s 93us/step - loss: 0.6926 - acc: 0.5216
Epoch 25/100
232/232 [==============================] - 0s 88us/step - loss: 0.6926 - acc: 0.5216
Epoch 26/100
232/232 [==============================] - 0s 89us/step - loss: 0.6926 - acc: 0.5216
Epoch 27/100
232/232 [==============================] - 0s 94us/step - loss: 0.6926 - acc: 0.5216
Epoch 28/100
232/232 [==============================] - 0s 97us/step - loss: 0.6925 - acc: 0.5216
Epoch 29/100
232/232 [==============================] - 0s 97us/step - loss: 0.6925 - acc: 0.5216
Epoch 30/100
232/232 [==============================] - 0s 91us/step - loss: 0.6925 - acc: 0.5216
Epoch 31/100
232/232 [==============================] - 0s 92us/step - loss: 0.6925 - acc: 0.5216
Epoch 32/100
232/232 [==============================] - 0s 95us/step - loss: 0.6925 - acc: 0.5216
Epoch 33/100
232/232 [==============================] - 0s 96us/step - loss: 0.6925 - acc: 0.5216
Epoch 34/100
232/232 [==============================] - 0s 91us/step - loss: 0.6925 - acc: 0.5216
Epoch 35/100
232/232 [==============================] - 0s 90us/step - loss: 0.6925 - acc: 0.5216
Epoch 36/100
232/232 [==============================] - 0s 93us/step - loss: 0.6925 - acc: 0.5216
Epoch 37/100
232/232 [==============================] - 0s 97us/step - loss: 0.6924 - acc: 0.5216
Epoch 38/100
232/232 [==============================] - 0s 86us/step - loss: 0.6925 - acc: 0.5216
```

```
Epoch 39/100
232/232 [==============================] - 0s 95us/step - loss: 0.6924 - acc: 0.5216
Epoch 40/100
232/232 [==============================] - 0s 95us/step - loss: 0.6924 - acc: 0.5216
Epoch 41/100
232/232 [==============================] - 0s 90us/step - loss: 0.6924 - acc: 0.5216
Epoch 42/100
232/232 [==============================] - 0s 95us/step - loss: 0.6924 - acc: 0.5216
Epoch 43/100
232/232 [==============================] - 0s 91us/step - loss: 0.6924 - acc: 0.5216
Epoch 44/100
232/232 [==============================] - 0s 91us/step - loss: 0.6924 - acc: 0.5216
Epoch 45/100
232/232 [==============================] - 0s 99us/step - loss: 0.6924 - acc: 0.5216
Epoch 46/100
232/232 [==============================] - 0s 95us/step - loss: 0.6924 - acc: 0.5216
Epoch 47/100
232/232 [==============================] - 0s 91us/step - loss: 0.6924 - acc: 0.5216
Epoch 48/100
232/232 [==============================] - 0s 89us/step - loss: 0.6924 - acc: 0.5216
Epoch 49/100
232/232 [==============================] - 0s 93us/step - loss: 0.6924 - acc: 0.5216
Epoch 50/100
232/232 [==============================] - 0s 95us/step - loss: 0.6924 - acc: 0.5216
Epoch 51/100
232/232 [==============================] - 0s 90us/step - loss: 0.6924 - acc: 0.5216
Epoch 52/100
232/232 [==============================] - 0s 89us/step - loss: 0.6923 - acc: 0.5216
Epoch 53/100
232/232 [==============================] - 0s 92us/step - loss: 0.6923 - acc: 0.5216
Epoch 54/100
232/232 [==============================] - 0s 91us/step - loss: 0.6924 - acc: 0.5216
Epoch 55/100
232/232 [==============================] - 0s 93us/step - loss: 0.6923 - acc: 0.5216
Epoch 56/100
232/232 [==============================] - 0s 97us/step - loss: 0.6923 - acc: 0.5216
Epoch 57/100
232/232 [==============================] - 0s 109us/step - loss: 0.6923 - acc: 0.5216
Epoch 58/100
232/232 [==============================] - 0s 102us/step - loss: 0.6923 - acc: 0.5216
Epoch 59/100
232/232 [==============================] - 0s 104us/step - loss: 0.6923 - acc: 0.5216
Epoch 60/100
232/232 [==============================] - 0s 91us/step - loss: 0.6923 - acc: 0.5216
Epoch 61/100
232/232 [==============================] - 0s 91us/step - loss: 0.6923 - acc: 0.5216
Epoch 62/100
232/232 [==============================] - 0s 91us/step - loss: 0.6923 - acc: 0.5216
Epoch 63/100
232/232 [==============================] - 0s 98us/step - loss: 0.6923 - acc: 0.5216
Epoch 64/100
232/232 [==============================] - 0s 93us/step - loss: 0.6923 - acc: 0.5216
Epoch 65/100
232/232 [==============================] - 0s 91us/step - loss: 0.6923 - acc: 0.5216
Epoch 66/100
232/232 [==============================] - 0s 94us/step - loss: 0.6923 - acc: 0.5216
Epoch 67/100
232/232 [==============================] - 0s 93us/step - loss: 0.6923 - acc: 0.5216
Epoch 68/100
232/232 [==============================] - 0s 90us/step - loss: 0.6923 - acc: 0.5216
Epoch 69/100
232/232 [==============================] - 0s 101us/step - loss: 0.6923 - acc: 0.5216
Epoch 70/100
232/232 [==============================] - 0s 91us/step - loss: 0.6923 - acc: 0.5216
Epoch 71/100
232/232 [==============================] - 0s 87us/step - loss: 0.6923 - acc: 0.5216
Epoch 72/100
232/232 [==============================] - 0s 90us/step - loss: 0.6923 - acc: 0.5216
Epoch 73/100
232/232 [==============================] - 0s 94us/step - loss: 0.6923 - acc: 0.5216
Epoch 74/100
232/232 [==============================] - 0s 93us/step - loss: 0.6923 - acc: 0.5216
Epoch 75/100
232/232 [==============================] - 0s 94us/step - loss: 0.6923 - acc: 0.5216
Epoch 76/100
232/232 [==============================] - 0s 90us/step - loss: 0.6923 - acc: 0.5216
Epoch 77/100
232/232 [==============================] - 0s 91us/step - loss: 0.6922 - acc: 0.5216
Epoch 78/100
232/232 [==============================] - 0s 93us/step - loss: 0.6922 - acc: 0.5216
Epoch 79/100
232/232 [==============================] - 0s 92us/step - loss: 0.6922 - acc: 0.5216
Epoch 80/100
232/232 [==============================] - 0s 90us/step - loss: 0.6922 - acc: 0.5216
Epoch 81/100
232/232 [==============================] - 0s 100us/step - loss: 0.6923 - acc: 0.5216
Epoch 82/100
232/232 [==============================] - 0s 95us/step - loss: 0.6922 - acc: 0.5216
Epoch 83/100
232/232 [==============================] - 0s 101us/step - loss: 0.6923 - acc: 0.5216
Epoch 84/100
232/232 [==============================] - 0s 95us/step - loss: 0.6923 - acc: 0.5216
Epoch 85/100
232/232 [==============================] - 0s 97us/step - loss: 0.6922 - acc: 0.5216
Epoch 86/100
```

```
232/232 [==============================] - 0s 90us/step - loss: 0.6922 - acc: 0.5216
Epoch 87/100
232/232 [==============================] - 0s 89us/step - loss: 0.6922 - acc: 0.5216
Epoch 88/100
232/232 [==============================] - 0s 99us/step - loss: 0.6923 - acc: 0.5216
Epoch 89/100
232/232 [==============================] - 0s 87us/step - loss: 0.6922 - acc: 0.5216
Epoch 90/100
232/232 [==============================] - 0s 100us/step - loss: 0.6922 - acc: 0.5216
Epoch 91/100
232/232 [==============================] - 0s 92us/step - loss: 0.6922 - acc: 0.5216
Epoch 92/100
232/232 [==============================] - 0s 90us/step - loss: 0.6922 - acc: 0.5216
Epoch 93/100
232/232 [==============================] - 0s 94us/step - loss: 0.6922 - acc: 0.5216
Epoch 94/100
232/232 [==============================] - 0s 93us/step - loss: 0.6922 - acc: 0.5216
Epoch 95/100
232/232 [==============================] - 0s 93us/step - loss: 0.6922 - acc: 0.5216
Epoch 96/100
232/232 [==============================] - 0s 92us/step - loss: 0.6922 - acc: 0.5216
Epoch 97/100
232/232 [==============================] - 0s 93us/step - loss: 0.6922 - acc: 0.5216
Epoch 98/100
232/232 [==============================] - 0s 93us/step - loss: 0.6922 - acc: 0.5216
Epoch 99/100
232/232 [==============================] - 0s 94us/step - loss: 0.6923 - acc: 0.5216
Epoch 100/100
232/232 [==============================] - 0s 94us/step - loss: 0.6922 - acc: 0.5216
116/116 [==============================] - 0s 644us/step
Epoch 1/100
232/232 [==============================] - 0s 1ms/step - loss: 0.6937 - acc: 0.5043
Epoch 2/100
232/232 [==============================] - 0s 95us/step - loss: 0.6928 - acc: 0.4957
Epoch 3/100
232/232 [==============================] - 0s 90us/step - loss: 0.6922 - acc: 0.4957
Epoch 4/100
232/232 [==============================] - 0s 96us/step - loss: 0.6913 - acc: 0.5647
Epoch 5/100
232/232 [==============================] - 0s 96us/step - loss: 0.6902 - acc: 0.5043
Epoch 6/100
232/232 [==============================] - 0s 94us/step - loss: 0.6889 - acc: 0.5043
Epoch 7/100
232/232 [==============================] - 0s 96us/step - loss: 0.6879 - acc: 0.5043
Epoch 8/100
232/232 [==============================] - 0s 91us/step - loss: 0.6847 - acc: 0.5043
Epoch 9/100
232/232 [==============================] - 0s 97us/step - loss: 0.6808 - acc: 0.5043
Epoch 10/100
232/232 [==============================] - 0s 97us/step - loss: 0.6744 - acc: 0.7328
Epoch 11/100
232/232 [==============================] - 0s 101us/step - loss: 0.6665 - acc: 0.8534
Epoch 12/100
232/232 [==============================] - 0s 94us/step - loss: 0.6560 - acc: 0.8362
Epoch 13/100
232/232 [==============================] - 0s 98us/step - loss: 0.6413 - acc: 0.6810
Epoch 14/100
232/232 [==============================] - 0s 95us/step - loss: 0.6263 - acc: 0.7931
Epoch 15/100
232/232 [==============================] - 0s 90us/step - loss: 0.6073 - acc: 0.7586
Epoch 16/100
232/232 [==============================] - 0s 96us/step - loss: 0.5797 - acc: 0.8621
Epoch 17/100
232/232 [==============================] - 0s 91us/step - loss: 0.5457 - acc: 0.8750
Epoch 18/100
232/232 [==============================] - 0s 93us/step - loss: 0.5150 - acc: 0.8922
Epoch 19/100
232/232 [==============================] - 0s 106us/step - loss: 0.4848 - acc: 0.8578
Epoch 20/100
232/232 [==============================] - 0s 95us/step - loss: 0.4422 - acc: 0.8836
Epoch 21/100
232/232 [==============================] - 0s 97us/step - loss: 0.4123 - acc: 0.9009
Epoch 22/100
232/232 [==============================] - 0s 100us/step - loss: 0.3846 - acc: 0.8922
Epoch 23/100
232/232 [==============================] - 0s 99us/step - loss: 0.3618 - acc: 0.8664
Epoch 24/100
232/232 [==============================] - 0s 93us/step - loss: 0.3222 - acc: 0.9138
Epoch 25/100
232/232 [==============================] - 0s 91us/step - loss: 0.3162 - acc: 0.9052
Epoch 26/100
232/232 [==============================] - 0s 95us/step - loss: 0.3027 - acc: 0.9009
Epoch 27/100
232/232 [==============================] - 0s 93us/step - loss: 0.2987 - acc: 0.8879
Epoch 28/100
232/232 [==============================] - 0s 95us/step - loss: 0.2531 - acc: 0.9267
Epoch 29/100
232/232 [==============================] - 0s 89us/step - loss: 0.2446 - acc: 0.9267
Epoch 30/100
232/232 [==============================] - 0s 92us/step - loss: 0.2445 - acc: 0.9267
Epoch 31/100
232/232 [==============================] - 0s 93us/step - loss: 0.2317 - acc: 0.9138
Epoch 32/100
232/232 [==============================] - 0s 90us/step - loss: 0.2128 - acc: 0.9310
Epoch 33/100
```

```
232/232 [==============================] - 0s 89us/step - loss: 0.2078 - acc: 0.9224
Epoch 34/100
232/232 [==============================] - 0s 98us/step - loss: 0.2132 - acc: 0.9310
Epoch 35/100
232/232 [==============================] - 0s 100us/step - loss: 0.1797 - acc: 0.9310
Epoch 36/100
232/232 [==============================] - 0s 90us/step - loss: 0.1910 - acc: 0.9224
Epoch 37/100
232/232 [==============================] - 0s 93us/step - loss: 0.1738 - acc: 0.9483
Epoch 38/100
232/232 [==============================] - 0s 100us/step - loss: 0.1768 - acc: 0.9353
Epoch 39/100
232/232 [==============================] - 0s 88us/step - loss: 0.1558 - acc: 0.9397
Epoch 40/100
232/232 [==============================] - 0s 89us/step - loss: 0.1484 - acc: 0.9569
Epoch 41/100
232/232 [==============================] - 0s 87us/step - loss: 0.1481 - acc: 0.9483
Epoch 42/100
232/232 [==============================] - 0s 90us/step - loss: 0.1383 - acc: 0.9569
Epoch 43/100
232/232 [==============================] - 0s 93us/step - loss: 0.1352 - acc: 0.9569
Epoch 44/100
232/232 [==============================] - 0s 97us/step - loss: 0.1337 - acc: 0.9612
Epoch 45/100
232/232 [==============================] - 0s 99us/step - loss: 0.1433 - acc: 0.9483
Epoch 46/100
232/232 [==============================] - 0s 99us/step - loss: 0.1218 - acc: 0.9569
Epoch 47/100
232/232 [==============================] - 0s 95us/step - loss: 0.1168 - acc: 0.9655
Epoch 48/100
232/232 [==============================] - 0s 96us/step - loss: 0.1185 - acc: 0.9526
Epoch 49/100
232/232 [==============================] - 0s 91us/step - loss: 0.1162 - acc: 0.9612
Epoch 50/100
232/232 [==============================] - 0s 90us/step - loss: 0.1121 - acc: 0.9569
Epoch 51/100
232/232 [==============================] - 0s 92us/step - loss: 0.1076 - acc: 0.9741
Epoch 52/100
232/232 [==============================] - 0s 96us/step - loss: 0.1145 - acc: 0.9698
Epoch 53/100
232/232 [==============================] - 0s 90us/step - loss: 0.1045 - acc: 0.9655
Epoch 54/100
232/232 [==============================] - 0s 98us/step - loss: 0.1002 - acc: 0.9698
Epoch 55/100
232/232 [==============================] - 0s 101us/step - loss: 0.0968 - acc: 0.9655
Epoch 56/100
232/232 [==============================] - 0s 96us/step - loss: 0.1004 - acc: 0.9741
Epoch 57/100
232/232 [==============================] - 0s 92us/step - loss: 0.0898 - acc: 0.9698
Epoch 58/100
232/232 [==============================] - 0s 92us/step - loss: 0.1045 - acc: 0.9526
Epoch 59/100
232/232 [==============================] - 0s 90us/step - loss: 0.0941 - acc: 0.9828
Epoch 60/100
232/232 [==============================] - 0s 93us/step - loss: 0.0878 - acc: 0.9741
Epoch 61/100
232/232 [==============================] - 0s 94us/step - loss: 0.0834 - acc: 0.9741
Epoch 62/100
232/232 [==============================] - 0s 93us/step - loss: 0.0911 - acc: 0.9741
Epoch 63/100
232/232 [==============================] - 0s 94us/step - loss: 0.0809 - acc: 0.9741
Epoch 64/100
232/232 [==============================] - 0s 103us/step - loss: 0.0854 - acc: 0.9741
Epoch 65/100
232/232 [==============================] - 0s 100us/step - loss: 0.0793 - acc: 0.9784
Epoch 66/100
232/232 [==============================] - 0s 97us/step - loss: 0.0765 - acc: 0.9784
Epoch 67/100
232/232 [==============================] - 0s 96us/step - loss: 0.0745 - acc: 0.9828
Epoch 68/100
232/232 [==============================] - 0s 96us/step - loss: 0.0743 - acc: 0.9741
Epoch 69/100
232/232 [==============================] - 0s 95us/step - loss: 0.0731 - acc: 0.9741
Epoch 70/100
232/232 [==============================] - 0s 93us/step - loss: 0.0806 - acc: 0.9698
Epoch 71/100
232/232 [==============================] - 0s 94us/step - loss: 0.0828 - acc: 0.9741
Epoch 72/100
232/232 [==============================] - 0s 93us/step - loss: 0.0847 - acc: 0.9741
Epoch 73/100
232/232 [==============================] - 0s 88us/step - loss: 0.0893 - acc: 0.9698
Epoch 74/100
232/232 [==============================] - 0s 88us/step - loss: 0.0675 - acc: 0.9828
Epoch 75/100
232/232 [==============================] - 0s 91us/step - loss: 0.0666 - acc: 0.9784
Epoch 76/100
232/232 [==============================] - 0s 91us/step - loss: 0.0697 - acc: 0.9828
Epoch 77/100
232/232 [==============================] - 0s 92us/step - loss: 0.0628 - acc: 0.9828
Epoch 78/100
232/232 [==============================] - 0s 101us/step - loss: 0.0626 - acc: 0.9784
Epoch 79/100
232/232 [==============================] - 0s 94us/step - loss: 0.0594 - acc: 0.9871
Epoch 80/100
232/232 [==============================] - 0s 91us/step - loss: 0.0633 - acc: 0.9828
```

```
Epoch 81/100
232/232 [==============================] - 0s 94us/step - loss: 0.0586 - acc: 0.9871
Epoch 82/100
232/232 [==============================] - 0s 95us/step - loss: 0.0563 - acc: 0.9871
Epoch 83/100
232/232 [==============================] - 0s 95us/step - loss: 0.0579 - acc: 0.9871
Epoch 84/100
232/232 [==============================] - 0s 95us/step - loss: 0.0551 - acc: 0.9871
Epoch 85/100
232/232 [==============================] - 0s 94us/step - loss: 0.0561 - acc: 0.9784
Epoch 86/100
232/232 [==============================] - 0s 96us/step - loss: 0.0735 - acc: 0.9655
Epoch 87/100
232/232 [==============================] - 0s 105us/step - loss: 0.0695 - acc: 0.9698
Epoch 88/100
232/232 [==============================] - 0s 96us/step - loss: 0.0691 - acc: 0.9698
Epoch 89/100
232/232 [==============================] - 0s 100us/step - loss: 0.0593 - acc: 0.9784
Epoch 90/100
232/232 [==============================] - 0s 94us/step - loss: 0.0562 - acc: 0.9871
Epoch 91/100
232/232 [==============================] - 0s 88us/step - loss: 0.0525 - acc: 0.9828
Epoch 92/100
232/232 [==============================] - 0s 89us/step - loss: 0.0526 - acc: 0.9828
Epoch 93/100
232/232 [==============================] - 0s 90us/step - loss: 0.0541 - acc: 0.9828
Epoch 94/100
232/232 [==============================] - 0s 92us/step - loss: 0.0464 - acc: 0.9871
Epoch 95/100
232/232 [==============================] - 0s 95us/step - loss: 0.0462 - acc: 0.9871
Epoch 96/100
232/232 [==============================] - 0s 88us/step - loss: 0.0463 - acc: 0.9914
Epoch 97/100
232/232 [==============================] - 0s 96us/step - loss: 0.0450 - acc: 0.9871
Epoch 98/100
232/232 [==============================] - 0s 97us/step - loss: 0.0431 - acc: 0.9914
Epoch 99/100
232/232 [==============================] - 0s 101us/step - loss: 0.0427 - acc: 0.9914
Epoch 100/100
232/232 [==============================] - 0s 91us/step - loss: 0.0433 - acc: 0.9914
116/116 [==============================] - 0s 653us/step
Accuracy mean: 0.7873563218390803
Accuracy variance: 0.24649076084024016
```