# SignLanguage

April 6, 2023

## 1 Sign Language Prediction with Deep Learning Models

*Nikhil Sharma*

**Objective : To learn how to implement deep learning models in a project with the help of Sign Language Dataset**

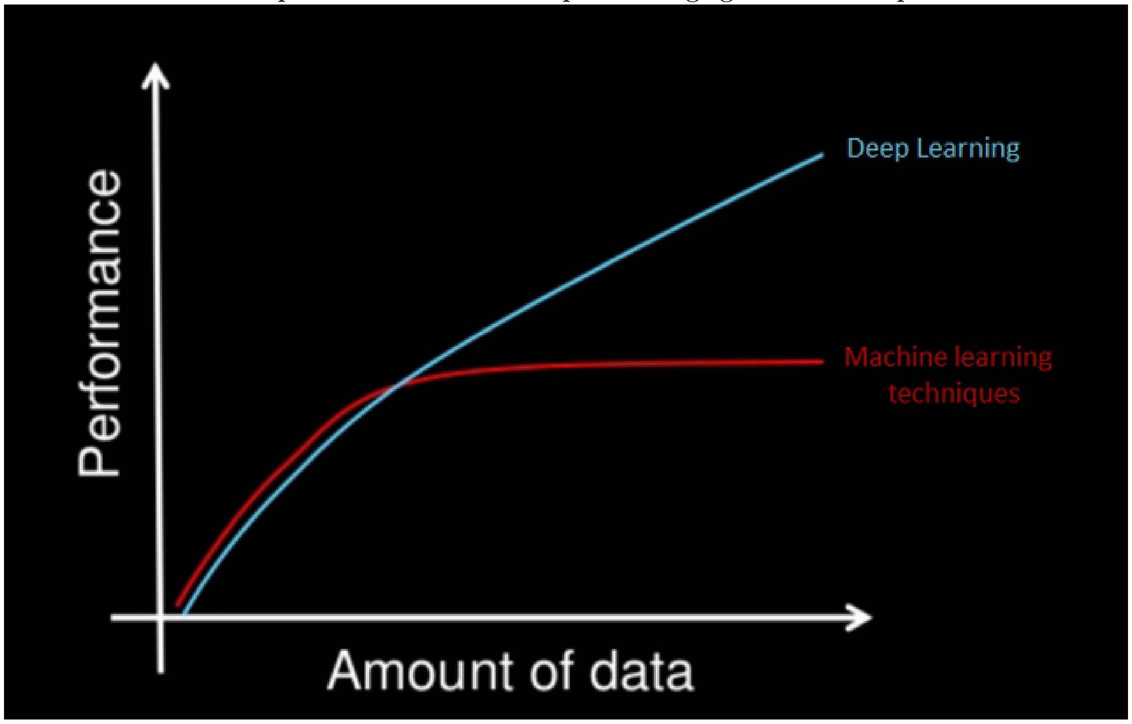
## 2 Table of Contents

Conclusion

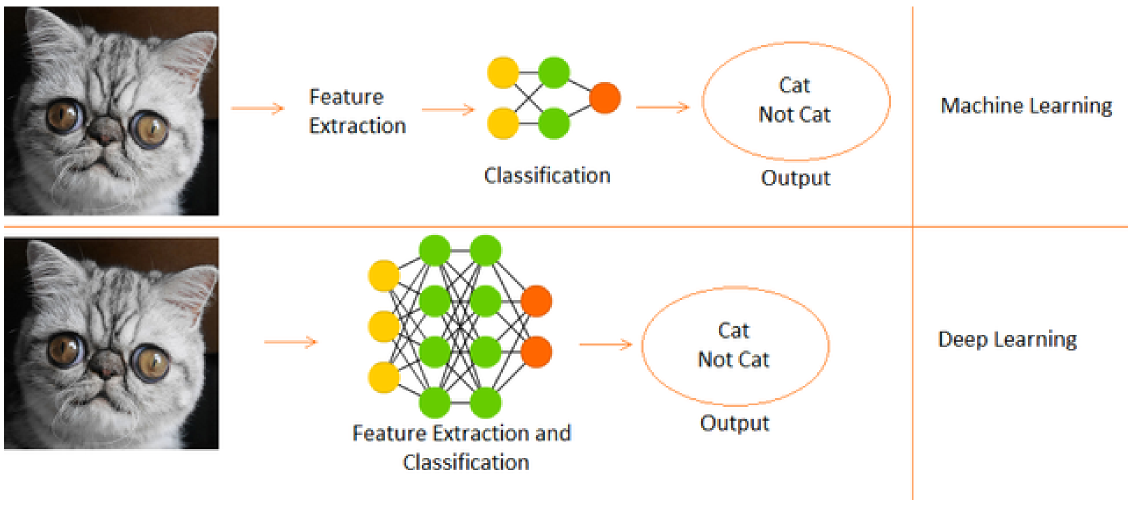
## 3 1. Introduction:

Sign languages (also known as signed languages) are languages that use manual communication to convey meaning. This can include simultaneously employing hand gestures, movement, orientation of the fingers, arms or body, and facial expressions to convey a speaker's ideas. Source: https://en.wikipedia.org/wiki/Sign_language

Deep learning, a state-of-the-art machine learning approach, has shown outstanding performance over traditional machine learning in identifying intricate structures in complex high-dimensional data, especially in the domain of computer vision. * **Deep learning:** One of the machine learning technique that learns features directly from data. * **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.



* **What is difference of deep learning from machine learning:** * Machine learning covers deep learning. * Features are given machine learning manually. * On the other hand, deep learning learns features directly from data.



## 4 2. Data Understanding

- We will use "sign language digits data set" for this tutorial.
- In this data there are 2062 sign language digits images.
- As you know digits are from 0 to 9. Therefore there are 10 unique sign.
- At the beginning of this project we will use only sign 0 and 1 for simplicity.
- In data, sign zero is between indexes 204 and 408. Number of zero sign is 205.
- Also sign one is between indexes 822 and 1027. Number of one sign is 206. Therefore, we will use 205 samples from each classes(labels).

**Note:** Actually 205 sample is very very very little for deep learning. But for this project it does not matter so much. Lets prepare our X and Y arrays. X is image array (zero and one signs) and Y is label array (0 and 1).

**Details of datasets:** * Image size: 64x64 * Color space: Grayscale * File format: npy * Number of classes: 10 (Digits: 0-9) * Number of participant students: 218 * Number of samples per student: 10

**Details of datasets in GitHub Repo:** * Repo: github.com/ardamavi/Sign-Language-Digits-Dataset

**Acknowledgements** *Sign Language Digits Dataset* * Dataset GitHub Page: github.com/ardamavi/Sign-Language-Digits-Dataset * By Turkey Ankara Ayrancı Anadolu High School Students * Turkey Ankara Ayrancı Anadolu High School's Sign Language Digits Dataset

## 5 3. Data Processing

- We will import the required libraries.
- We will Load the both training and testing dataset.

## 5.1 3.1 Importing Libraries

[1]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt


# import warnings
import warnings
# filter warnings
warnings.filterwarnings('ignore')
```

## 5.2 3.2 Data Loading

[2]:
```
# load data set
x_df = np.load('data/X.npy')
Y_df = np.load('data/Y.npy')
img_size = 64
plt.subplot(1, 2, 1)
plt.imshow(x_df[111].reshape(img_size, img_size))
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(x_df[344].reshape(img_size, img_size))
plt.axis('off')
```

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

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

[3]:
```
# Join a sequence of arrays along an row axis.
```

```
X = np.concatenate((x_df[204:409],x_df[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 represent 410 images in the dataset(images of sign language for zero's and one's).
  - 64 represents the size of our image (i.e. 64x64 pixels)
- The shape of the Y is (410,1)
  - 410 represent that we have 410 labels (0's and 1's)
- 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.

## 5.3 3.3 Data Splitting

[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 it into 2D array 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).

[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)
```

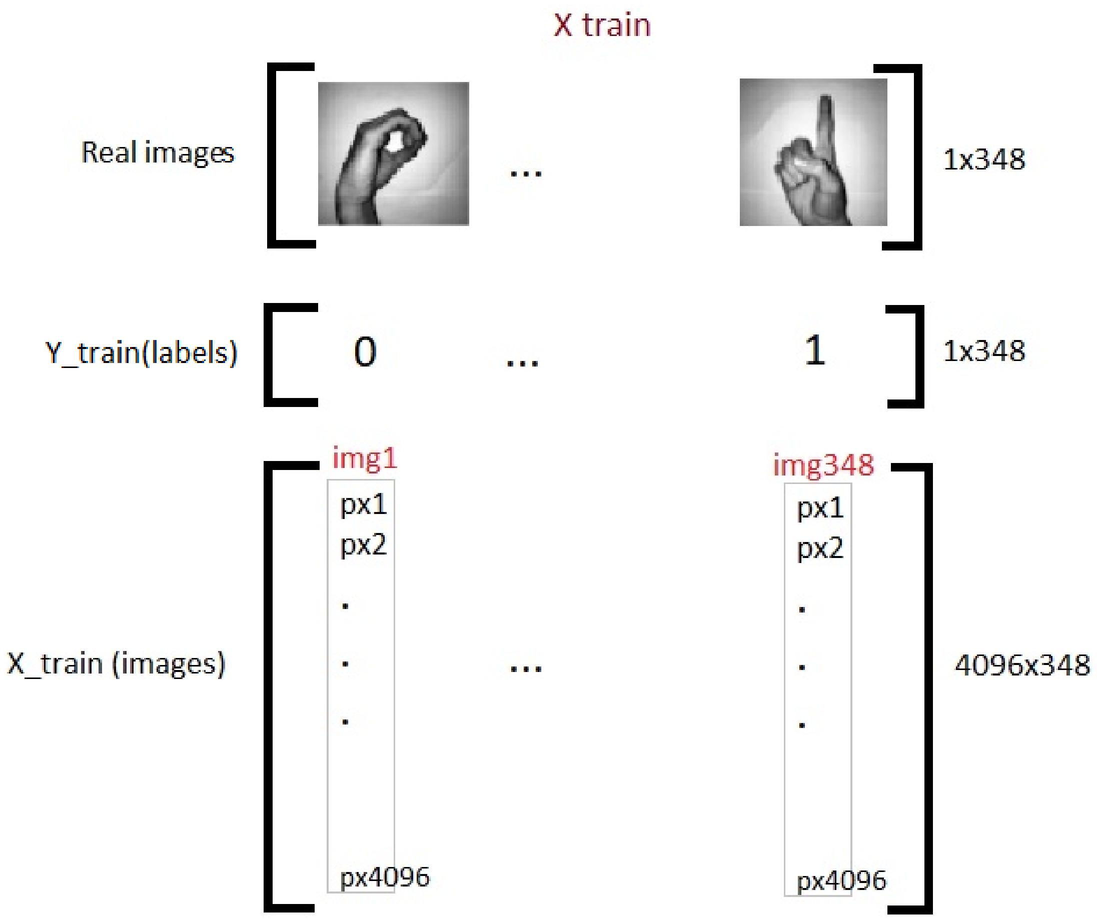
- As you can see, we have 348 images and each image has 4096 pixels in image train array.
- Also, we have 62 images and each image has 4096 pixels in image test array.
- Then lets take transpose.

```
[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



## 6 4. Logistic Regression

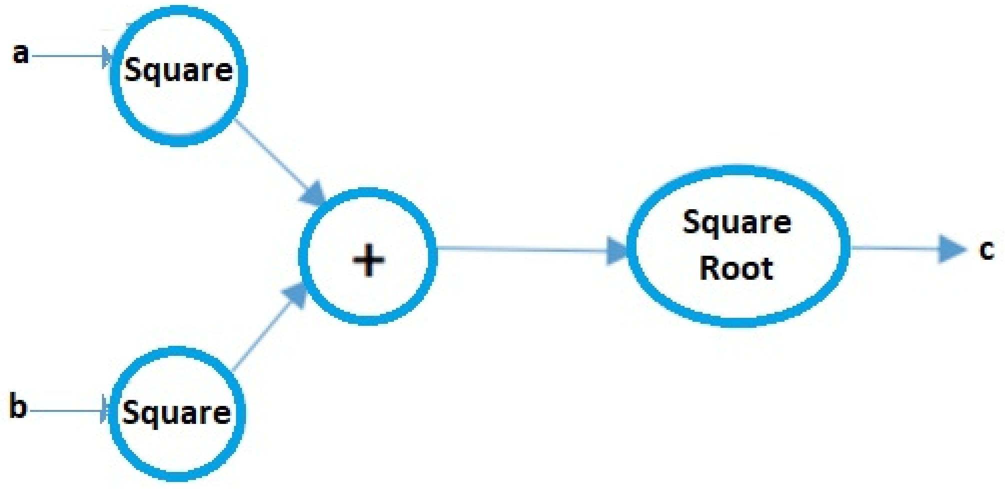
- When we talk about binary classification( 0 and 1 outputs) what comes to mind first is logistic regression.
- However, in deep learning project, what to do with logistic regression there?
  - The answer is that logistic regression is actually a very simple neural network.
- In order to understand logistic regression (simple deep learning) lets first learn computation graph.

### 6.1 4.1 Computation Graph

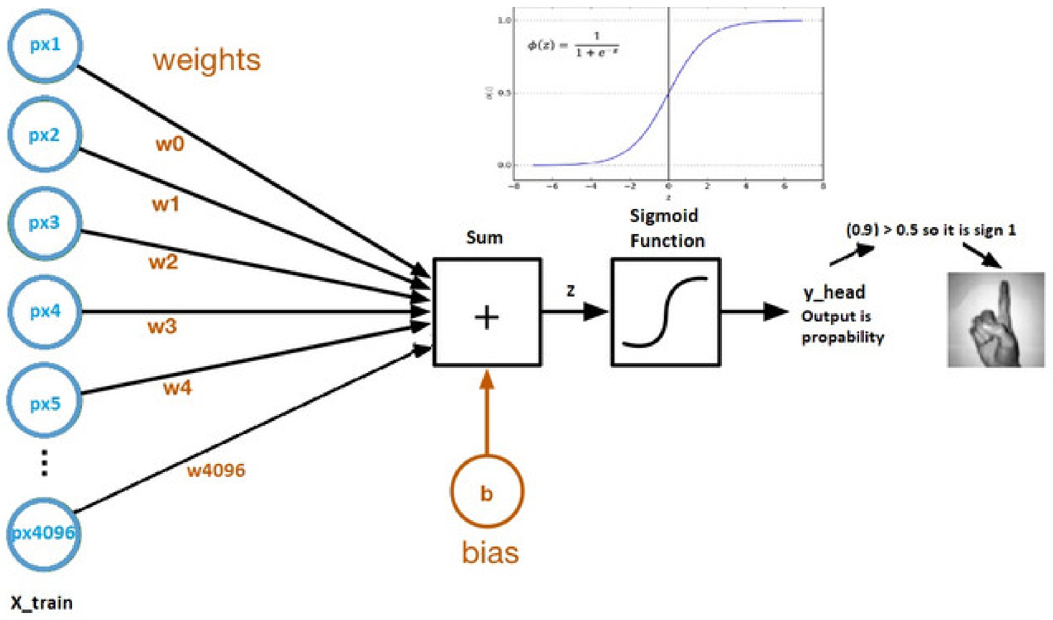
- Computation graphs are a nice way to think about mathematical expressions.
- It is like visualization of mathematical expressions.For example we have:

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

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



- Now lets look at 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 + px1w1 + px2w2 + … + 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 (we will see as soon.)
- 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 at artificial neural network but for this time initial weights are 0.01. * 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), I make definition(method).

[7]:
```
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
```

### 6.2 4.2 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)

[8]:
```
# calculation of z
#z = np.dot(w.T,x_train)+b
def sigmoid(z):
    y_head = 1/(1+np.exp(-z))
    return y_head
```

[9]:
```
y_head = sigmoid(0)
y_head
```

[9]:
```
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:

$$-(1 - y)log(1 - yhat) - ylogyhat$$

  - 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.

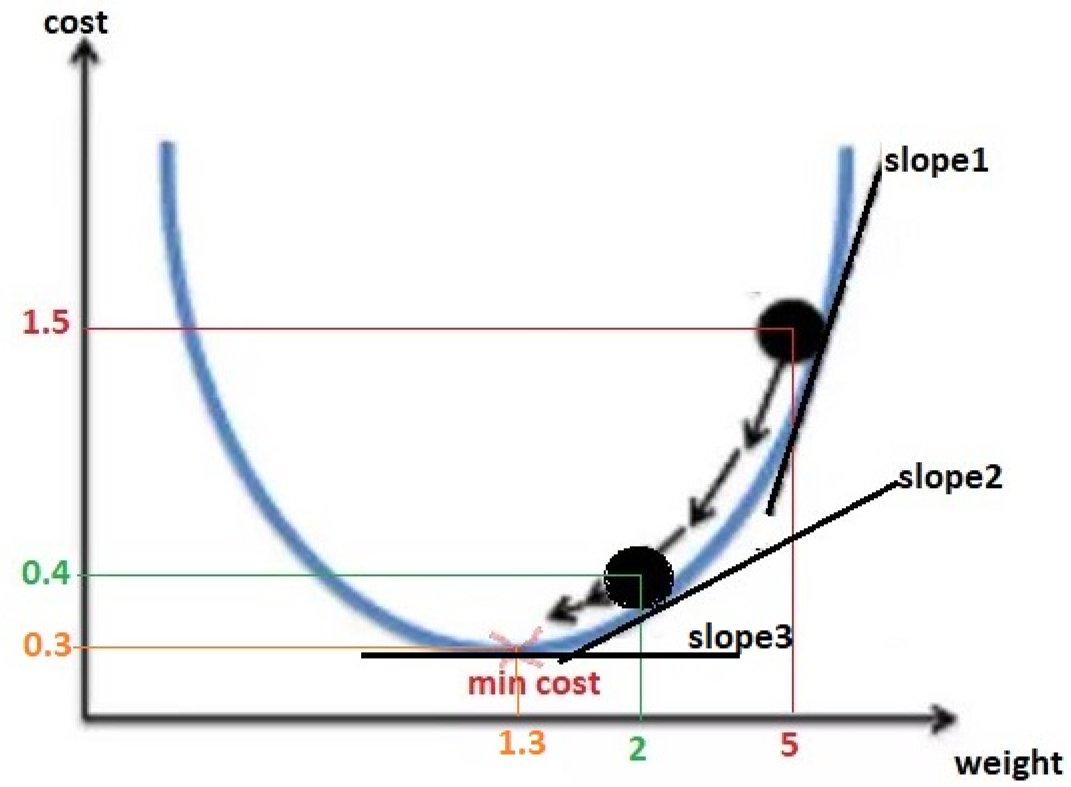
- After that, the cost function is summation of loss function. Each image creates loss function. Cost function is summation of loss functions that is created by each input image.
- Lets implement forward propagation.

[10]:
```
# 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
```

## 6.3 4.3 Backward Propagation

**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 ou
```



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 $. 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. 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. * Updated equation is following. 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 $\alpha$ learning rate. Then update weight. (In order to explain I ignore

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

bias but these all steps will be applied for bias) * 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 the 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.

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

$$ { J\over b} = {1 \over m} { _{i=1}^m(y^{head}-y)} $$

```
[11]: # 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.

```
[12]: # 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)
```

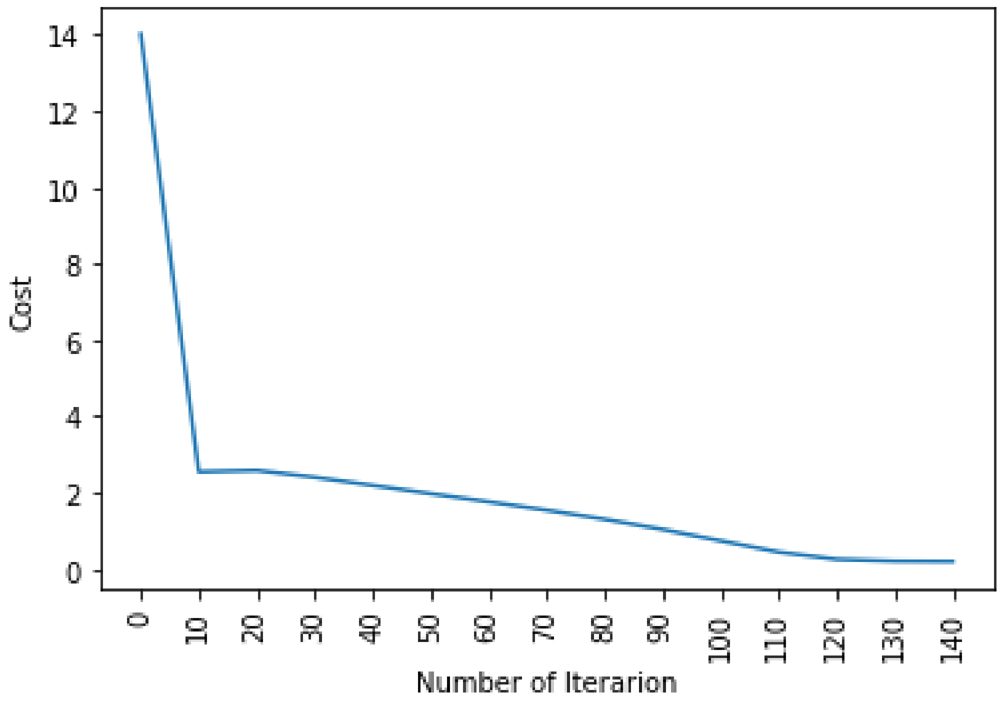
- Up to this point we learn our parameters. It means we fit the data.
- In order to predict we have parameters. Therefore, lets predict.
- In prediction step we have x_test as a input and while using it, we make forward prediction.

[13]:
```
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.

[14]:
```
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.

### 6.4 4.4 Logistic Regression with Sklearn

- In sklearn library, there is a logistic regression method that ease implementing logistic regression.
- I am not going to explain each parameter of logistic regression in sklearn but if you want you can read from there http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- 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.

[15]:
```
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
```

**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

## 7 5 Artificial Neural Network

- 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.

### 7.1 5.1 Two-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)}: $

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

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

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

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

- 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.

[16]:
```
# 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
```

### 7.2 5.2 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.

[17]:
```
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
```

### 7.3 5.3 Loss function and Cost function

- Loss and cost functions are same with logistic regression
- Cross entropy function

$$J(\theta) = -\sum_{i} y_i ln(y_i^{(hat)})$$

[18]:
```
# 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
```

### 7.4 5.4 Backward Propagation

- As you know backward propagation means derivative.

[19]:
```
# 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. * We actually do alot of work with logistic regression

[20]:
```
# 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.

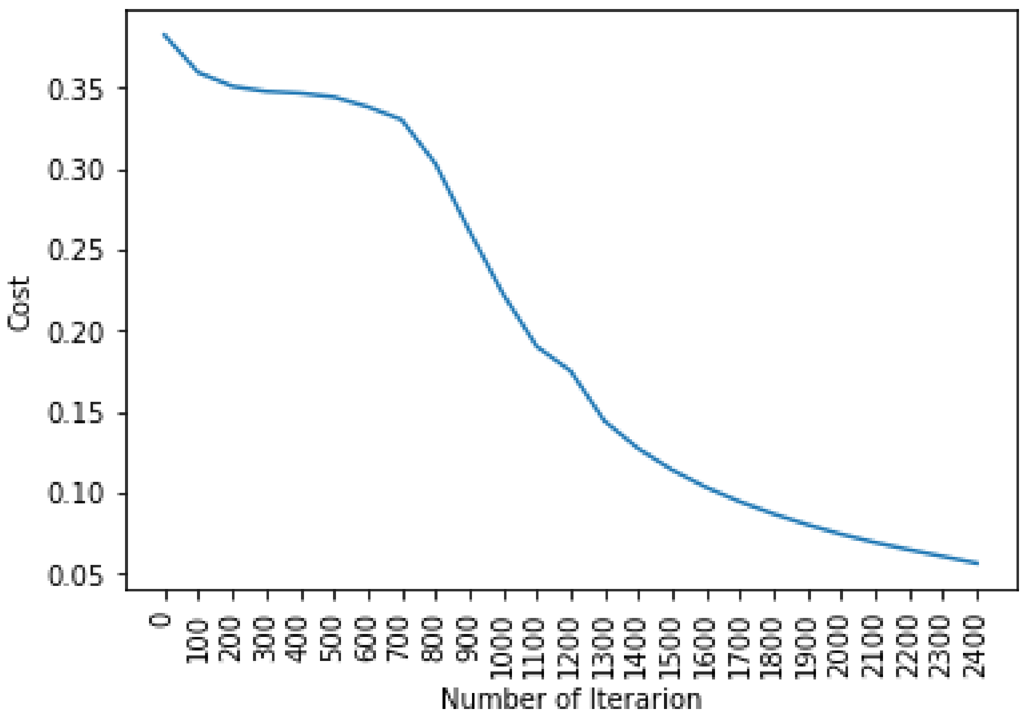
[21]:
```python
# prediction
def predict_NN(parameters,x_test):
    # x_test is a input for forward propagation
    A2, cache = forward_propagation_NN(x_test,parameters)
    Y_prediction = np.zeros((1,x_test.shape[1]))
    # if z is bigger than 0.5, our prediction is sign one (y_head=1),
    # if z is 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
```

## 7.5 5.5 Create Model

- Lets put them all together

[22]:
```python
# 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: 0.382568
Cost after iteration 100: 0.359447
Cost after iteration 200: 0.351083
Cost after iteration 300: 0.347938
Cost after iteration 400: 0.346853
Cost after iteration 500: 0.344510
Cost after iteration 600: 0.338493
Cost after iteration 700: 0.330476
Cost after iteration 800: 0.303413
Cost after iteration 900: 0.261983
Cost after iteration 1000: 0.222920
Cost after iteration 1100: 0.190346
Cost after iteration 1200: 0.175218
Cost after iteration 1300: 0.144352
Cost after iteration 1400: 0.127283
Cost after iteration 1500: 0.114121
Cost after iteration 1600: 0.103412
Cost after iteration 1700: 0.094522
Cost after iteration 1800: 0.086939
Cost after iteration 1900: 0.080274
Cost after iteration 2000: 0.074392
Cost after iteration 2100: 0.069301
Cost after iteration 2200: 0.064822
Cost after iteration 2300: 0.060670
Cost after iteration 2400: 0.056517
```



```
train accuracy: 99.13793103448276 %
test accuracy: 93.54838709677419 %
```

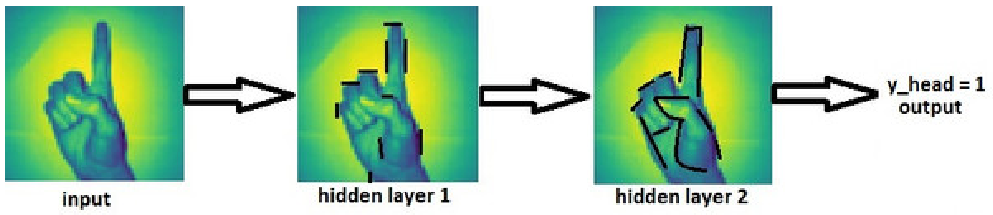
Up to this point we create 2 layer neural network

- 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

Now lets implement L layer neural network with keras.

## 8 6. L Layer Neural Network

- What happens if number of hidden layer increase: Earlier layerls 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.
  - These hyperparameters can be chosen intuitively if you spend a lot of time in deep learning world.
  - In this project our model will have 2 hidden layer with 8 and 4 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.

[23]:
```
# reshaping
x_train, x_test, y_train, y_test = x_train.T, x_test.T, y_train.T, y_test.T
```

## 8.1 6.1 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.

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

- metrics: it is accuracy.
- cross_val_score: use cross validation. If you do not know cross validation please chech it from my machine learning tutorial. https://www.kaggle.com/kanncaa1/machine-learning-tutorial-for-beginners
- epochs: number of iteration

[24]:
```
# Evaluating the ANN
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
```

```python
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))
```

```
Epoch 1/100
8/8 [==============================] - 1s 5ms/step - loss: 0.6929 - accuracy:
0.5216
Epoch 2/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6917 - accuracy:
0.5431
Epoch 3/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6907 - accuracy:
0.5431
Epoch 4/100
8/8 [==============================] - 0s 2ms/step - loss: 0.6891 - accuracy:
0.5431
Epoch 5/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6882 - accuracy:
0.5431
Epoch 6/100
8/8 [==============================] - 0s 4ms/step - loss: 0.6870 - accuracy:
0.5431
Epoch 7/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6861 - accuracy:
0.5431
Epoch 8/100
8/8 [==============================] - 0s 4ms/step - loss: 0.6856 - accuracy:
0.5431
Epoch 9/100
8/8 [==============================] - 0s 4ms/step - loss: 0.6849 - accuracy:
```

```
0.5431
Epoch 10/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6835 - accuracy:
0.5431
Epoch 11/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6823 - accuracy:
0.5431
Epoch 12/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6778 - accuracy:
0.5431
Epoch 13/100
8/8 [==============================] - 0s 4ms/step - loss: 0.6747 - accuracy:
0.6207
Epoch 14/100
8/8 [==============================] - 0s 8ms/step - loss: 0.6681 - accuracy:
0.6207
Epoch 15/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6607 - accuracy:
0.5991
Epoch 16/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6498 - accuracy:
0.8147
Epoch 17/100
8/8 [==============================] - 0s 4ms/step - loss: 0.6330 - accuracy:
0.7414
Epoch 18/100
8/8 [==============================] - 0s 4ms/step - loss: 0.6142 - accuracy:
0.8190
Epoch 19/100
8/8 [==============================] - 0s 4ms/step - loss: 0.5903 - accuracy:
0.7586
Epoch 20/100
8/8 [==============================] - 0s 5ms/step - loss: 0.5826 - accuracy:
0.7888
Epoch 21/100
8/8 [==============================] - 0s 5ms/step - loss: 0.5386 - accuracy:
0.8448
Epoch 22/100
8/8 [==============================] - 0s 5ms/step - loss: 0.5144 - accuracy:
0.9052
Epoch 23/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4815 - accuracy:
0.9052
Epoch 24/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4534 - accuracy:
0.8750
Epoch 25/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4366 - accuracy:
```

```
0.9009
Epoch 26/100
8/8 [==============================] - 0s 4ms/step - loss: 0.4108 - accuracy:
0.8793
Epoch 27/100
8/8 [==============================] - 0s 2ms/step - loss: 0.3770 - accuracy:
0.9009
Epoch 28/100
8/8 [==============================] - 0s 3ms/step - loss: 0.3616 - accuracy:
0.9052
Epoch 29/100
8/8 [==============================] - 0s 3ms/step - loss: 0.3393 - accuracy:
0.9009
Epoch 30/100
8/8 [==============================] - 0s 3ms/step - loss: 0.3124 - accuracy:
0.9224
Epoch 31/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2958 - accuracy:
0.9095
Epoch 32/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2839 - accuracy:
0.9224
Epoch 33/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2729 - accuracy:
0.9138
Epoch 34/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2723 - accuracy:
0.9095
Epoch 35/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2832 - accuracy:
0.8879
Epoch 36/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2338 - accuracy:
0.9224
Epoch 37/100
8/8 [==============================] - 0s 4ms/step - loss: 0.2349 - accuracy:
0.9267
Epoch 38/100
8/8 [==============================] - 0s 4ms/step - loss: 0.2258 - accuracy:
0.9267
Epoch 39/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2177 - accuracy:
0.9181
Epoch 40/100
8/8 [==============================] - 0s 5ms/step - loss: 0.2166 - accuracy:
0.9397
Epoch 41/100
8/8 [==============================] - 0s 4ms/step - loss: 0.2185 - accuracy:
```

```
0.9310
Epoch 42/100
8/8 [==============================] - 0s 4ms/step - loss: 0.2018 - accuracy:
0.9397
Epoch 43/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2314 - accuracy:
0.9052
Epoch 44/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2633 - accuracy:
0.9009
Epoch 45/100
8/8 [==============================] - 0s 4ms/step - loss: 0.2690 - accuracy:
0.9009
Epoch 46/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2141 - accuracy:
0.9353
Epoch 47/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2001 - accuracy:
0.9138
Epoch 48/100
8/8 [==============================] - 0s 5ms/step - loss: 0.1748 - accuracy:
0.9483
Epoch 49/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1703 - accuracy:
0.9397
Epoch 50/100
8/8 [==============================] - 0s 4ms/step - loss: 0.1763 - accuracy:
0.9267
Epoch 51/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1674 - accuracy:
0.9267
Epoch 52/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1750 - accuracy:
0.9353
Epoch 53/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1767 - accuracy:
0.9483
Epoch 54/100
8/8 [==============================] - 0s 4ms/step - loss: 0.1814 - accuracy:
0.9310
Epoch 55/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1544 - accuracy:
0.9397
Epoch 56/100
8/8 [==============================] - 0s 1ms/step - loss: 0.1568 - accuracy:
0.9310
Epoch 57/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1585 - accuracy:
```

```
0.9569
Epoch 58/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1513 - accuracy:
0.9353
Epoch 59/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1479 - accuracy:
0.9397
Epoch 60/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1443 - accuracy:
0.9397
Epoch 61/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1487 - accuracy:
0.9526
Epoch 62/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1395 - accuracy:
0.9397
Epoch 63/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1360 - accuracy:
0.9483
Epoch 64/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1384 - accuracy:
0.9526
Epoch 65/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1386 - accuracy:
0.9440
Epoch 66/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1330 - accuracy:
0.9397
Epoch 67/100
8/8 [==============================] - 0s 1ms/step - loss: 0.1405 - accuracy:
0.9569
Epoch 68/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1328 - accuracy:
0.9440
Epoch 69/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1265 - accuracy:
0.9526
Epoch 70/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1270 - accuracy:
0.9526
Epoch 71/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1207 - accuracy:
0.9483
Epoch 72/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1234 - accuracy:
0.9526
Epoch 73/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1240 - accuracy:
```

```
0.9483
Epoch 74/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1193 - accuracy:
0.9569
Epoch 75/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1145 - accuracy:
0.9569
Epoch 76/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1162 - accuracy:
0.9569
Epoch 77/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1133 - accuracy:
0.9483
Epoch 78/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1167 - accuracy:
0.9612
Epoch 79/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1198 - accuracy:
0.9483
Epoch 80/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1118 - accuracy:
0.9612
Epoch 81/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1119 - accuracy:
0.9526
Epoch 82/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1049 - accuracy:
0.9612
Epoch 83/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1067 - accuracy:
0.9483
Epoch 84/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1031 - accuracy:
0.9655
Epoch 85/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1101 - accuracy:
0.9569
Epoch 86/100
8/8 [==============================] - 0s 1ms/step - loss: 0.1004 - accuracy:
0.9655
Epoch 87/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1089 - accuracy:
0.9483
Epoch 88/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1035 - accuracy:
0.9698
Epoch 89/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1049 - accuracy:
```

```
0.9483
Epoch 90/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1141 - accuracy:
0.9655
Epoch 91/100
8/8 [==============================] - 0s 1ms/step - loss: 0.1022 - accuracy:
0.9569
Epoch 92/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0942 - accuracy:
0.9698
Epoch 93/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0926 - accuracy:
0.9655
Epoch 94/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0932 - accuracy:
0.9655
Epoch 95/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0891 - accuracy:
0.9655
Epoch 96/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0876 - accuracy:
0.9655
Epoch 97/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0920 - accuracy:
0.9784
Epoch 98/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0885 - accuracy:
0.9612
Epoch 99/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0846 - accuracy:
0.9741
Epoch 100/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0881 - accuracy:
0.9569
4/4 [==============================] - 0s 2ms/step - loss: 0.0888 - accuracy:
0.9741
Epoch 1/100
8/8 [==============================] - 1s 2ms/step - loss: 0.6933 - accuracy:
0.4828
Epoch 2/100
8/8 [==============================] - 0s 2ms/step - loss: 0.6927 - accuracy:
0.6552
Epoch 3/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6921 - accuracy:
0.4784
Epoch 4/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6915 - accuracy:
0.4784
```

```
Epoch 5/100
8/8 [==============================] - 0s 2ms/step - loss: 0.6887 - accuracy:
0.4914
Epoch 6/100
8/8 [==============================] - 0s 1ms/step - loss: 0.6859 - accuracy:
0.5819
Epoch 7/100
8/8 [==============================] - 0s 2ms/step - loss: 0.6794 - accuracy:
0.5216
Epoch 8/100
8/8 [==============================] - 0s 2ms/step - loss: 0.6705 - accuracy:
0.5862
Epoch 9/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6637 - accuracy:
0.6940
Epoch 10/100
8/8 [==============================] - 0s 2ms/step - loss: 0.6483 - accuracy:
0.6724
Epoch 11/100
8/8 [==============================] - 0s 2ms/step - loss: 0.6401 - accuracy:
0.5388
Epoch 12/100
8/8 [==============================] - 0s 2ms/step - loss: 0.6209 - accuracy:
0.8017
Epoch 13/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5986 - accuracy:
0.5647
Epoch 14/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6047 - accuracy:
0.8405
Epoch 15/100
8/8 [==============================] - 0s 2ms/step - loss: 0.5585 - accuracy:
0.7371
Epoch 16/100
8/8 [==============================] - 0s 2ms/step - loss: 0.5370 - accuracy:
0.8190
Epoch 17/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5227 - accuracy:
0.8578
Epoch 18/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4967 - accuracy:
0.7931
Epoch 19/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4787 - accuracy:
0.8664
Epoch 20/100
8/8 [==============================] - 0s 2ms/step - loss: 0.4678 - accuracy:
0.8578
```

```
Epoch 21/100
8/8 [==============================] - 0s 2ms/step - loss: 0.4568 - accuracy:
0.8491
Epoch 22/100
8/8 [==============================] - 0s 2ms/step - loss: 0.4371 - accuracy:
0.9181
Epoch 23/100
8/8 [==============================] - 0s 2ms/step - loss: 0.4246 - accuracy:
0.9181
Epoch 24/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4094 - accuracy:
0.8664
Epoch 25/100
8/8 [==============================] - 0s 2ms/step - loss: 0.4182 - accuracy:
0.9397
Epoch 26/100
8/8 [==============================] - 0s 2ms/step - loss: 0.4025 - accuracy:
0.9095
Epoch 27/100
8/8 [==============================] - 0s 2ms/step - loss: 0.3969 - accuracy:
0.9224
Epoch 28/100
8/8 [==============================] - 0s 2ms/step - loss: 0.3692 - accuracy:
0.9612
Epoch 29/100
8/8 [==============================] - 0s 2ms/step - loss: 0.3607 - accuracy:
0.9310
Epoch 30/100
8/8 [==============================] - 0s 2ms/step - loss: 0.3691 - accuracy:
0.9698
Epoch 31/100
8/8 [==============================] - 0s 3ms/step - loss: 0.3666 - accuracy:
0.9009
Epoch 32/100
8/8 [==============================] - 0s 2ms/step - loss: 0.3608 - accuracy:
0.9267
Epoch 33/100
8/8 [==============================] - 0s 4ms/step - loss: 0.3713 - accuracy:
0.9440
Epoch 34/100
8/8 [==============================] - 0s 2ms/step - loss: 0.3478 - accuracy:
0.9483
Epoch 35/100
8/8 [==============================] - 0s 2ms/step - loss: 0.3258 - accuracy:
0.9440
Epoch 36/100
8/8 [==============================] - 0s 3ms/step - loss: 0.3227 - accuracy:
0.9569
```

```
Epoch 37/100
8/8 [==============================] - 0s 2ms/step - loss: 0.3083 - accuracy:
0.9483
Epoch 38/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2990 - accuracy:
0.9698
Epoch 39/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2937 - accuracy:
0.9569
Epoch 40/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2917 - accuracy:
0.9612
Epoch 41/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2791 - accuracy:
0.9784
Epoch 42/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2813 - accuracy:
0.9483
Epoch 43/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2696 - accuracy:
0.9655
Epoch 44/100
8/8 [==============================] - 0s 2ms/step - loss: 0.3074 - accuracy:
0.9526
Epoch 45/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2761 - accuracy:
0.9655
Epoch 46/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2752 - accuracy:
0.9741
Epoch 47/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2576 - accuracy:
0.9526
Epoch 48/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2376 - accuracy:
0.9741
Epoch 49/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2308 - accuracy:
0.9784
Epoch 50/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2203 - accuracy:
0.9828
Epoch 51/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2144 - accuracy:
0.9741
Epoch 52/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2200 - accuracy:
0.9784
```

```
Epoch 53/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2008 - accuracy:
0.9828
Epoch 54/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1928 - accuracy:
0.9828
Epoch 55/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1989 - accuracy:
0.9741
Epoch 56/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1937 - accuracy:
0.9784
Epoch 57/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1828 - accuracy:
0.9828
Epoch 58/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1745 - accuracy:
0.9828
Epoch 59/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1729 - accuracy:
0.9871
Epoch 60/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1672 - accuracy:
0.9784
Epoch 61/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1681 - accuracy:
0.9914
Epoch 62/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2109 - accuracy:
0.9526
Epoch 63/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2088 - accuracy:
0.9526
Epoch 64/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1537 - accuracy:
0.9957
Epoch 65/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1522 - accuracy:
0.9828
Epoch 66/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1410 - accuracy:
0.9828
Epoch 67/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1371 - accuracy:
0.9828
Epoch 68/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1388 - accuracy:
0.9871
```

```
Epoch 69/100
8/8 [==============================] - 0s 1ms/step - loss: 0.1327 - accuracy:
0.9871
Epoch 70/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1274 - accuracy:
0.9828
Epoch 71/100
8/8 [==============================] - 0s 1ms/step - loss: 0.1242 - accuracy:
0.9871
Epoch 72/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1263 - accuracy:
0.9914
Epoch 73/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1217 - accuracy:
0.9914
Epoch 74/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1215 - accuracy:
0.9914
Epoch 75/100
8/8 [==============================] - 0s 1ms/step - loss: 0.1107 - accuracy:
0.9914
Epoch 76/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1076 - accuracy:
0.9914
Epoch 77/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1208 - accuracy:
0.9871
Epoch 78/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1437 - accuracy:
0.9655
Epoch 79/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1328 - accuracy:
0.9828
Epoch 80/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1081 - accuracy:
0.9914
Epoch 81/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0976 - accuracy:
0.9957
Epoch 82/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1085 - accuracy:
0.9828
Epoch 83/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0973 - accuracy:
0.9957
Epoch 84/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0942 - accuracy:
0.9914
```

```
Epoch 85/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1035 - accuracy:
0.9828
Epoch 86/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0907 - accuracy:
0.9957
Epoch 87/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0783 - accuracy:
0.9957
Epoch 88/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0777 - accuracy:
0.9957
Epoch 89/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0764 - accuracy:
1.0000
Epoch 90/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0719 - accuracy:
1.0000
Epoch 91/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0732 - accuracy:
0.9957
Epoch 92/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0717 - accuracy:
1.0000
Epoch 93/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0686 - accuracy:
0.9957
Epoch 94/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0657 - accuracy:
0.9957
Epoch 95/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0628 - accuracy:
1.0000
Epoch 96/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0607 - accuracy:
1.0000
Epoch 97/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0603 - accuracy:
1.0000
Epoch 98/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0586 - accuracy:
1.0000
Epoch 99/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0575 - accuracy:
1.0000
Epoch 100/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0545 - accuracy:
1.0000
```

```
4/4 [==============================] - 0s 6ms/step - loss: 0.1708 - accuracy:
0.9397
Epoch 1/100
8/8 [==============================] - 1s 4ms/step - loss: 0.6931 - accuracy:
0.5043
Epoch 2/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6921 - accuracy:
0.5043
Epoch 3/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6910 - accuracy:
0.5043
Epoch 4/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6901 - accuracy:
0.5043
Epoch 5/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6877 - accuracy:
0.5043
Epoch 6/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6852 - accuracy:
0.5302
Epoch 7/100
8/8 [==============================] - 0s 2ms/step - loss: 0.6793 - accuracy:
0.5129
Epoch 8/100
8/8 [==============================] - 0s 2ms/step - loss: 0.6706 - accuracy:
0.5086
Epoch 9/100
8/8 [==============================] - 0s 2ms/step - loss: 0.6589 - accuracy:
0.5259
Epoch 10/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6490 - accuracy:
0.6552
Epoch 11/100
8/8 [==============================] - 0s 2ms/step - loss: 0.6259 - accuracy:
0.7069
Epoch 12/100
8/8 [==============================] - 0s 1ms/step - loss: 0.6160 - accuracy:
0.6724
Epoch 13/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5880 - accuracy:
0.6638
Epoch 14/100
8/8 [==============================] - 0s 2ms/step - loss: 0.5524 - accuracy:
0.7759
Epoch 15/100
8/8 [==============================] - 0s 2ms/step - loss: 0.5255 - accuracy:
0.8276
Epoch 16/100
```

```
8/8 [==============================] - 0s 2ms/step - loss: 0.4896 - accuracy:
0.8534
Epoch 17/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4623 - accuracy:
0.8448
Epoch 18/100
8/8 [==============================] - 0s 1ms/step - loss: 0.4068 - accuracy:
0.8836
Epoch 19/100
8/8 [==============================] - 0s 3ms/step - loss: 0.3560 - accuracy:
0.9009
Epoch 20/100
8/8 [==============================] - 0s 3ms/step - loss: 0.3210 - accuracy:
0.9138
Epoch 21/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2869 - accuracy:
0.9095
Epoch 22/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2648 - accuracy:
0.9052
Epoch 23/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2456 - accuracy:
0.9310
Epoch 24/100
8/8 [==============================] - 0s 2ms/step - loss: 0.2350 - accuracy:
0.9353
Epoch 25/100
8/8 [==============================] - 0s 3ms/step - loss: 0.2045 - accuracy:
0.9397
Epoch 26/100
8/8 [==============================] - 0s 1ms/step - loss: 0.2130 - accuracy:
0.9224
Epoch 27/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1758 - accuracy:
0.9440
Epoch 28/100
8/8 [==============================] - 0s 1ms/step - loss: 0.1860 - accuracy:
0.9267
Epoch 29/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1667 - accuracy:
0.9440
Epoch 30/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1620 - accuracy:
0.9526
Epoch 31/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1468 - accuracy:
0.9569
Epoch 32/100
```

```
8/8 [==============================] - 0s 2ms/step - loss: 0.1406 - accuracy:
0.9569
Epoch 33/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1411 - accuracy:
0.9440
Epoch 34/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1360 - accuracy:
0.9569
Epoch 35/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1191 - accuracy:
0.9655
Epoch 36/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1129 - accuracy:
0.9569
Epoch 37/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1099 - accuracy:
0.9655
Epoch 38/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1093 - accuracy:
0.9655
Epoch 39/100
8/8 [==============================] - 0s 3ms/step - loss: 0.1114 - accuracy:
0.9698
Epoch 40/100
8/8 [==============================] - 0s 2ms/step - loss: 0.1124 - accuracy:
0.9612
Epoch 41/100
8/8 [==============================] - 0s 4ms/step - loss: 0.0926 - accuracy:
0.9741
Epoch 42/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0953 - accuracy:
0.9655
Epoch 43/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0894 - accuracy:
0.9784
Epoch 44/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0849 - accuracy:
0.9784
Epoch 45/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0839 - accuracy:
0.9784
Epoch 46/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0804 - accuracy:
0.9828
Epoch 47/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0881 - accuracy:
0.9655
Epoch 48/100
```

```
8/8 [==============================] - 0s 3ms/step - loss: 0.0831 - accuracy:
0.9784
Epoch 49/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0712 - accuracy:
0.9828
Epoch 50/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0881 - accuracy:
0.9741
Epoch 51/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0716 - accuracy:
0.9784
Epoch 52/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0693 - accuracy:
0.9828
Epoch 53/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0673 - accuracy:
0.9828
Epoch 54/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0651 - accuracy:
0.9828
Epoch 55/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0748 - accuracy:
0.9741
Epoch 56/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0821 - accuracy:
0.9612
Epoch 57/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0771 - accuracy:
0.9784
Epoch 58/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0823 - accuracy:
0.9655
Epoch 59/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0621 - accuracy:
0.9828
Epoch 60/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0589 - accuracy:
0.9828
Epoch 61/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0654 - accuracy:
0.9741
Epoch 62/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0710 - accuracy:
0.9698
Epoch 63/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0657 - accuracy:
0.9784
Epoch 64/100
```

```
8/8 [==============================] - 0s 1ms/step - loss: 0.0712 - accuracy:
0.9741
Epoch 65/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0575 - accuracy:
0.9828
Epoch 66/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0547 - accuracy:
0.9871
Epoch 67/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0442 - accuracy:
0.9914
Epoch 68/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0741 - accuracy:
0.9655
Epoch 69/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0932 - accuracy:
0.9655
Epoch 70/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0622 - accuracy:
0.9828
Epoch 71/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0536 - accuracy:
0.9828
Epoch 72/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0424 - accuracy:
0.9914
Epoch 73/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0422 - accuracy:
0.9871
Epoch 74/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0478 - accuracy:
0.9871
Epoch 75/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0404 - accuracy:
0.9914
Epoch 76/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0519 - accuracy:
0.9828
Epoch 77/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0449 - accuracy:
0.9828
Epoch 78/100
8/8 [==============================] - 0s 1ms/step - loss: 0.0353 - accuracy:
0.9957
Epoch 79/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0449 - accuracy:
0.9871
Epoch 80/100
```

```
8/8 [==============================] - 0s 2ms/step - loss: 0.0522 - accuracy:
0.9871
Epoch 81/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0998 - accuracy:
0.9526
Epoch 82/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0394 - accuracy:
0.9914
Epoch 83/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0488 - accuracy:
0.9871
Epoch 84/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0406 - accuracy:
0.9871
Epoch 85/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0417 - accuracy:
0.9828
Epoch 86/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0367 - accuracy:
0.9871
Epoch 87/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0317 - accuracy:
0.9957
Epoch 88/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0292 - accuracy:
0.9914
Epoch 89/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0303 - accuracy:
1.0000
Epoch 90/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0365 - accuracy:
0.9871
Epoch 91/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0346 - accuracy:
0.9828
Epoch 92/100
8/8 [==============================] - 0s 6ms/step - loss: 0.0297 - accuracy:
0.9957
Epoch 93/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0326 - accuracy:
0.9957
Epoch 94/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0467 - accuracy:
0.9784
Epoch 95/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0304 - accuracy:
0.9914
Epoch 96/100
```

```
8/8 [==============================] - 0s 2ms/step - loss: 0.0313 - accuracy:
0.9957
Epoch 97/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0337 - accuracy:
0.9871
Epoch 98/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0269 - accuracy:
0.9914
Epoch 99/100
8/8 [==============================] - 0s 2ms/step - loss: 0.0226 - accuracy:
0.9957
Epoch 100/100
8/8 [==============================] - 0s 3ms/step - loss: 0.0229 - accuracy:
0.9914
4/4 [==============================] - 0s 4ms/step - loss: 0.1454 - accuracy:
0.9397
Accuracy mean: 0.9511494239171346
Accuracy variance: 0.01625530892527764
```

**Artificial Neural Network with Pytorch library.** * Pytorch is one of the frame works like keras. * It eases implementing and constructing deep learning blocks. * Artificial Neural Network: https://www.kaggle.com/kanncaa1/pytorch-tutorial-for-deep-learning-lovers

**Convolutional Neural Network with Pytorch library.** * Pytorch is one of the frame works like keras. * It eases implementing and constructing deep learning blocks. * Convolutional Neural Network: https://www.kaggle.com/kanncaa1/pytorch-tutorial-for-deep-learning-lovers

**Recurrent Neural Network with Pytorch library.** * Pytorch is one of the frame works like keras. * It eases implementing and constructing deep learning blocks. * Recurrent Neural Network: https://www.kaggle.com/kanncaa1/recurrent-neural-network-with-pytorch

## 8.2 7. Conclusion

- First of all thanks for this data set.
- I got to learn how to implement Deep Learning models on a project
- I cleared my doubts regarding parameter tuning, backward propagation, and different deep learning topics.
- The activation method was harder for me but as I completed this project the topic was very easy to comprehend.