Lab Session-IV

(Logistic Regression Classifier as a Perceptron Neural Network)

DR. JASMEET SINGH,
ASSISTANT PROFESSOR,
CSED, TIET

Logistic Regression as Neural Network

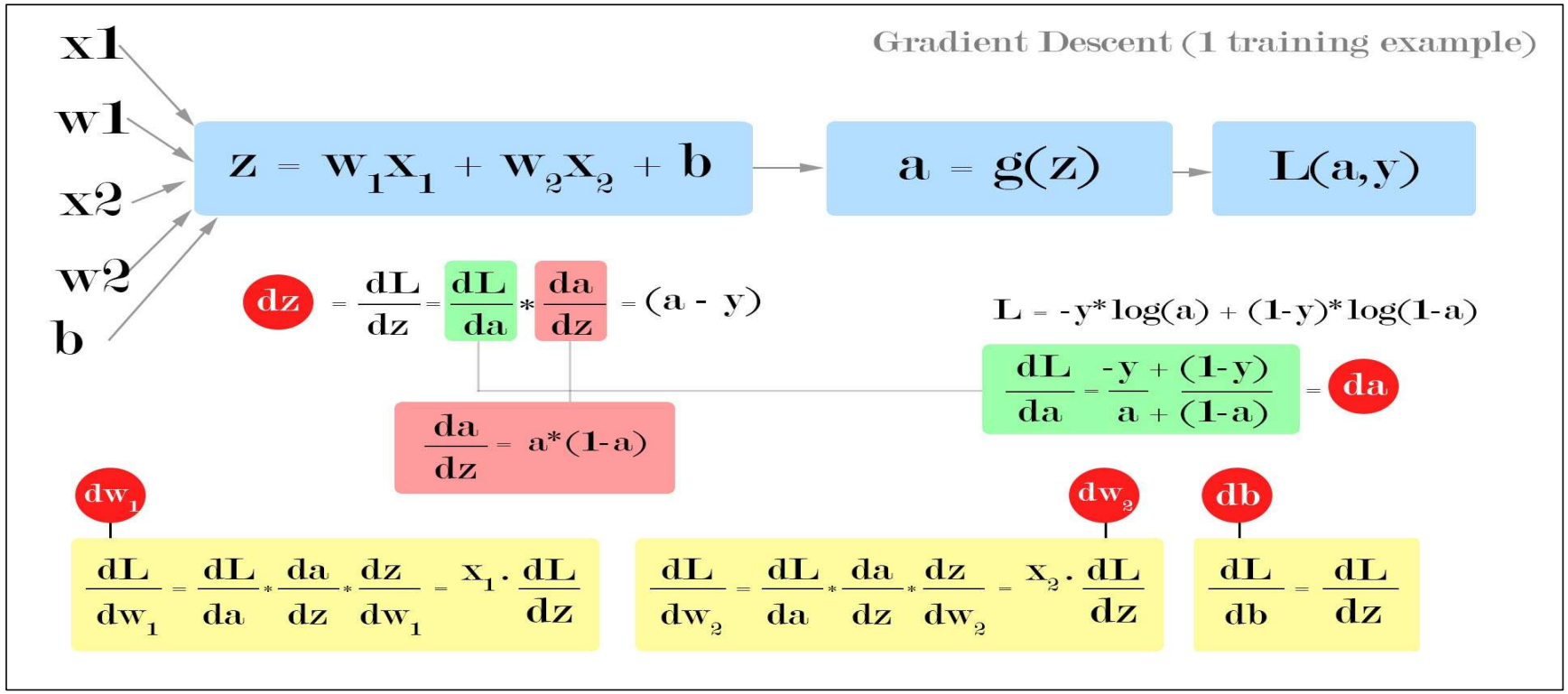


Image Classification using NN-based Logistic Regression Classifier

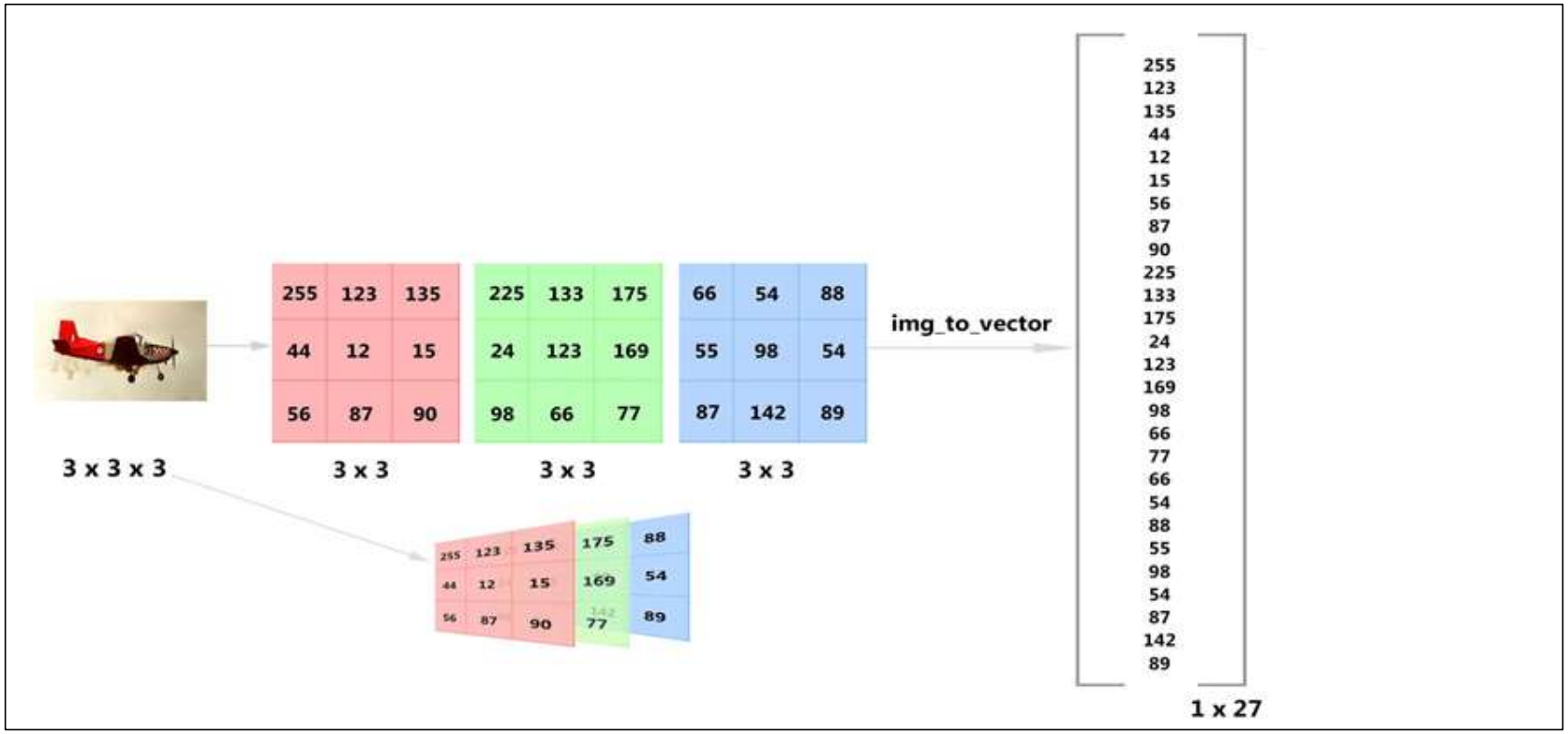
- We will use the CALTECH-101 dataset which has images belonging to 101 categories such as airplane, bike, elephant etc. As we are dealing with a binary classification problem, we will specifically use images from two categories airplane and bike.
- There are 800 images of both airplane and bike, which are divided into 750 and 50 for training and testing respectively
  - dataset -> train -> airplane -> 750 images

    dataset -> train -> bike -> 750 images

    dataset -> test -> airplane -> 50 images

    dataset -> test -> bike -> 50 images

Load the dataset



Load the dataset-code

```
import numpy as np
import os
from keras.preprocessing import image
train_path1='path to training plane files'
train_path2=' path to training bike files'
test_path1='path to test plane files'
test_path2='path to test bike files'
```

```
#Initializing training and test arrays
train_x = np.zeros((12288,1500),dtype=np.float)
train_y = np.zeros((1,1500),dtype=np.float)
test_x = np.zeros((12288,98),dtype=np.float)
test_y = np.zeros((1,98),dtype=np.float)
```

Load the dataset-code (Contd….)

```
#Loading training images in training arrays
train_files1=os.listdir(train_path1)
train_files2=os.listdir(train_path2)
for i in range(len(train_files1)):
  img = image.load_img(train_path1+train_files1[i], target_size=(64,64))
  arr=np.array(img)
  arr=arr.flatten()
  train_x[:,i]=arr
  train_y[0,i]=0
for i in range(len(train_files2)):
  img = image.load_img(train_path2+train_files2[i], target_size=(64,64))
  arr=np.array(img)
  arr=arr.flatten()
  train_x[:,i+750]=arr
  train_y[0,i+750]=1
```

Load the dataset-code (Contd….)

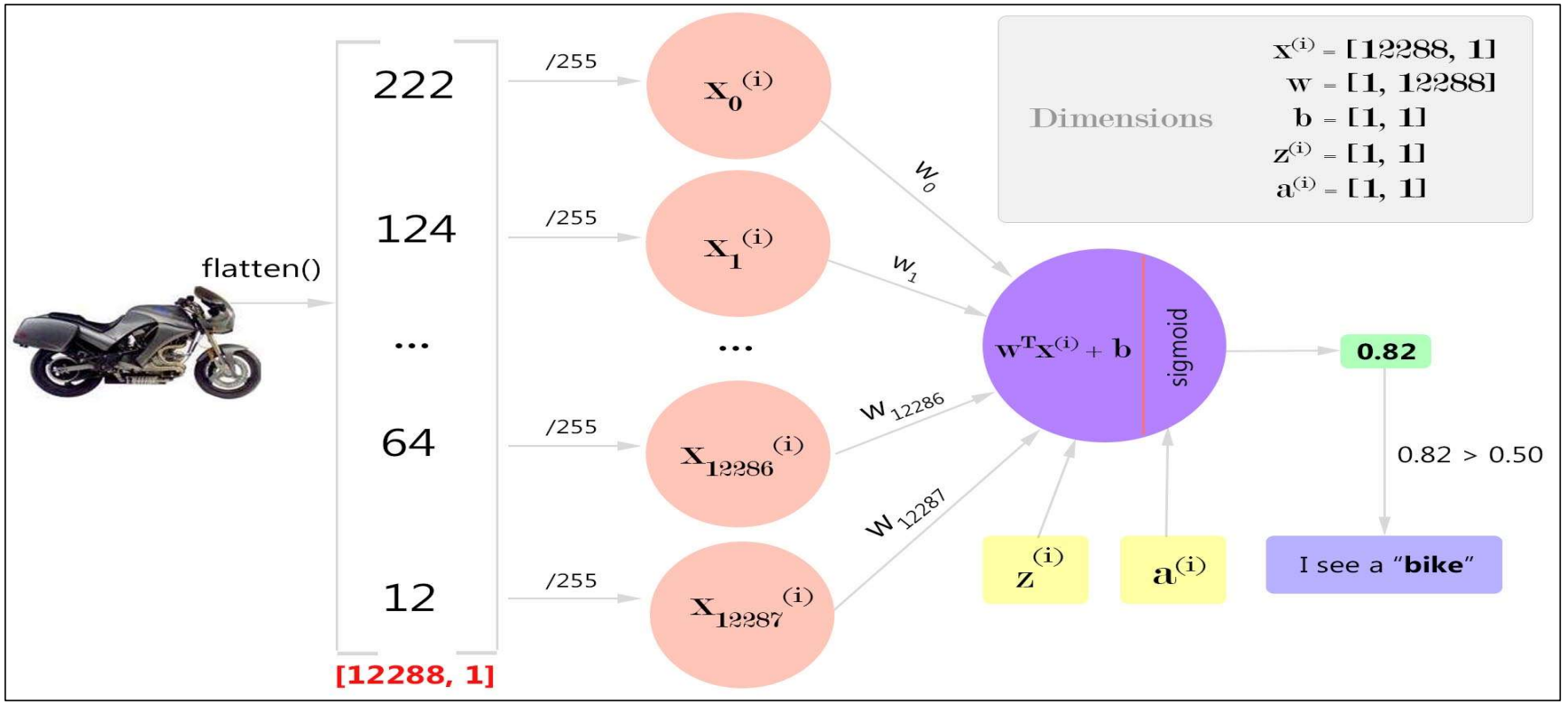
```
#Loading testing images in testing arrays
test_files1=os.listdir(test_path1)
test_files2=os.listdir(test_path2)
for i in range(len(test_files1)):
  img = image.load_img(test_path1+test_files1[i], target_size=(64,64))
  arr=np.array(img)
  arr=arr.flatten()
  test_x[:,i]=arr
  test_y[0,i]=0
for i in range(len(test_files2)):
  img = image.load_img(test_path2+test_files2[i], target_size=(64,64))
  arr=np.array(img)
  arr=arr.flatten()
  train_x[:,i+51]=arr
  train_y[0,i+51]=1
```

Load the dataset-code (Contd….)

Normalizing the train and test matrix:

```
train_x = train_x/255.
test_x  = test_x/255.
```

Logistic Regression Pipleine



Code- Sigmoid and init_param functions

sigmoid()

- *Input* - a number or a numpy array.
- *Output* - sigmoid of the number or the numpy array.

```
def sigmoid(z):
    return (1/(1+np.exp(-z)))
```

init_params()

- *Input* - dimension for weights (every value in an image's vector has a weight associated with it).
- *Output* - weight vector w and bias b

```
def init_params(dimension):
    w = np.zeros((dimension,1))
    b = 0
    return w, b
```

Code- Propagation function

`propagate()`

- *Input* - weight vector `w`, bias `b`, image matrix `X` and label vector `Y`.
- *Output* - gradients `dw`, `db` and cost function `costs` for every 10 iterations.
  - Forward propagation (for a single training example)
    - Calculate the weighted sum $z = w_1 x_1 + w_2 x_2 + b$.
    - Calculate the activation $a = \sigma(z)$.
    - Compute the loss $L(a, y) = -ylog(a) + (1 - y)log(1 - a)$.
  - Backpropagation (for a single training example)
    - Compute the derivatives of parameters $\frac{dL}{dw1}$, $\frac{dL}{dw2}$ and $\frac{dL}{db}$ using $\frac{dL}{da}$ and $\frac{dL}{dz}$.
    - Use update rule to update the parameters.
      - $w1 = w1 - \alpha \frac{dL}{dw1}$
      - $w2 = w2 - \alpha \frac{dL}{dw2}$
      - $b = b - \alpha \frac{dL}{db}$

Code- Propagation function

```
def propagate(w, b, X, Y):
   # num of training samples
   m = X.shape[1]
   # forward pass
   predicted     = sigmoid(np.dot(w.T,X) + b)
   cost = (-1/m)*(np.sum(np.multiply(Y,np.log(predicted)) + np.multiply((1-
Y),np.log(1-predicted))))
  # back propagation
   dw = (1/m)*(np.dot(X, (predicted-Y).T))
   db = (1/m)*(np.sum(predicted-Y))

  cost = np.squeeze(cost)

  # gradient dictionary
   grads = {"dw": dw, "db": db}

  return grads, cost
```

Code- Optimize function

`optimize()`

- *Input* - weight vector `w`, bias `b`, image matrix `X`, label vector `Y`, number of iterations for gradient descent `epochs` and learning rate `lr`.
- *Output* - parameter dictionary `params` holding updated `w` and `b`, gradient dictionary `grads` holding `dw` and `db`, and list of cost function `costs` after every 100 iterations.

Code- Optimize function

```
def optimize(w, b, X, Y, epochs, lr):
    costs = []
    for i in range(epochs):
        # calculate gradients
        grads, cost = propagate(w, b, X, Y)
        # get gradients
        dw = grads["dw"]
        db = grads["db"]
        # update rule
        w = w - (lr*dw)
        b = b - (lr*db)
        if i % 10 == 0:
            costs.append(cost)
            print("cost after %i epochs: %f" %(i, cost))
    # param dict
    params = {"w": w, "b": b}
    # gradient dict
    grads  = {"dw": dw, "db": db}

    return params, grads, costs
```

Code-Predict Function

`predict()`

- *Input* - updated parameters `w`, `b` and image matrix `X`.
- *Output* - predicted labels `Y_predict` for the image matrix `X`

Code-Predict Function

```
def predict(w, b, X):

  m = X.shape[1]

  Y_predict = np.zeros((1,m))
 A = sigmoid(np.dot(w.T, X) + b)
  for i in range(A.shape[1]):

    if A[0, i] <= 0.5:

      Y_predict[0, i] = 0

    else:

      Y_predict[0,i]  = 1


  return Y_predict
```

Code-Combining all modules

```
def model(X_train, Y_train, X_test, Y_test, epochs, lr):
  w, b = init_params(X_train.shape[0])
  params, grads, costs = optimize(w, b, X_train, Y_train, epochs, lr)

  w = params["w"]
  b = params["b"]

  Y_predict_train = predict(w, b, X_train)
  Y_predict_test  = predict(w, b, X_test)

  print("train_accuracy: {} %".format(100-np.mean(np.abs(Y_predict_train - Y_train)) * 100))
  print("test_accuracy : {} %".format(100-np.mean(np.abs(Y_predict_test  - Y_test)) * 100))
model(train_x, train_y, test_x, test_y, 100, 0.001)
```