Animals classification with Logistic Regression and Neural Networks

Gasan Nazer, Veronika Yankova
Complutense University of Madrid

1. Introduction

This document describes all steps of the process of creating our final project for the subject “Machine learning and big data”. It covers all decisions we took, from choosing a suitable dataset and preprocessing it, to creating different machine learning algorithms, their implementations, and evaluation. Finally, you will see the results we have obtained and our conclusions.

2. Dataset

As a requirement of the given assignment, we had to use a dataset from the website Kaggle. After downloading and checking several sets, we chose to use Animals-10. It includes 10 types of animals with sets from 1000 to over 5000 images per type. Another requirement of our project was to resolve a classification problem and a set like this fits well for the task. After sharing our dataset with our professor, he agreed on it and additionally suggested we use not all ten animals but just three - to reduce computational time.

Preprocessing

After downloading the dataset we started cleaning it for the first time- manually. Later, after implementing some of our models, we realized it will be better if we automate this process, as much as possible.

The preprocessing of our dataset consists of unzipping the downloaded “Animals-10” set and changing folders’ names - translating them to English. Additionally, we had to change the format of all images to RGB, as most of the images were already in it as “jpg” and models with dimensions of images (width, height, 3) are more common. We have converted some of them from RGB-A - “png”, others were in Grayscale and we have deleted them (as there was only one, we thought that it will not have any negative effect, but simplifies the process).
We also detected that there are big differences, in numbers of images, between the three types we have chosen - cats, dogs, and elephants. For elephants, we had twice fewer images than for cats and four times less than dogs. We decided to reduce the number of images with cats and dogs so that we can have a more equilibrated dataset. This decision was made mainly because the algorithms were predicting everything as dogs since the probability of the animal being a dog was way higher than the probability of being any other.
At this moment, our dataset was ready to be used but we needed to separate it into three parts- train, cross-validation (dev), and test with the corresponding proportions 80%, 10%, 10%. For the training dataset, after equilibrating, we have 1120 images of the three types of animals. For the dev and test dataset, after equilibrating, we have 140 images of the three types of animals, each.

Next, we needed to load all datasets and construct our input vectors “X” and output labels “Y”. For that, we created several helper functions, that load and reshape out input to the dimensions we later use (num_px, num_px, 3)- where num_px is the new width/height of our images and “3” are the number of color channels in RGB. During this process, we also generate our labels. As they were not given in a separate file, we needed to create them during loading time. Having our images separated into folders by type of animal we easily were able to map all images to the corresponding One-Hot Encodings- “dog” → “100”; “cat” → “010”; “elephant” → “001”.

For more information regarding the code and how to execute it (see Appendix A).

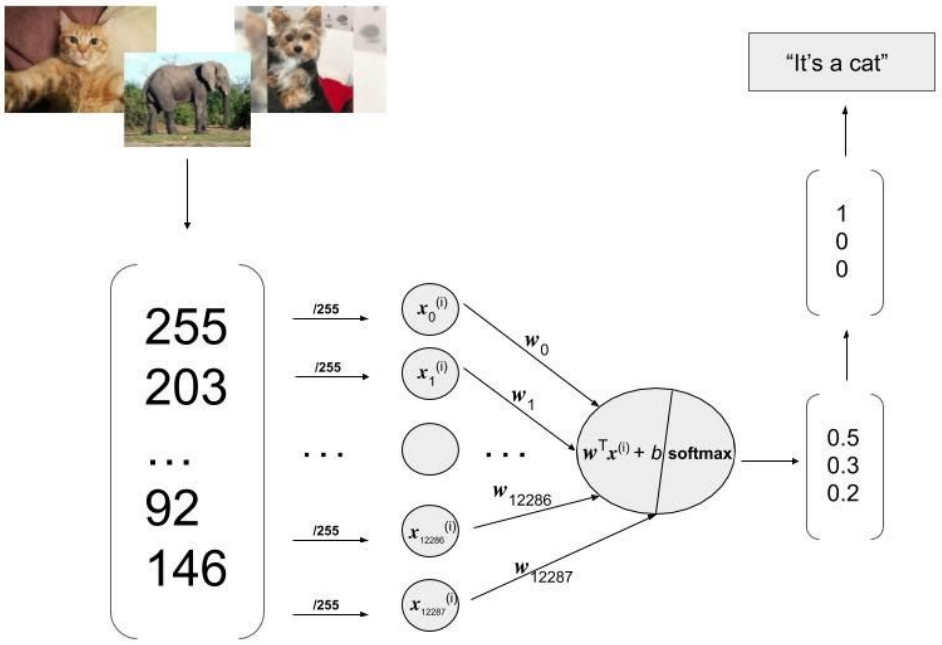
3. Models

As we have to create a model capable of predicting whether an image of a cat, a dog, or an elephant is the corresponding animal, we are facing a multi-class (multinomial) problem. To solve it, we have created several different models grouped in logistic regression models and neural network models.

Logistic regression models

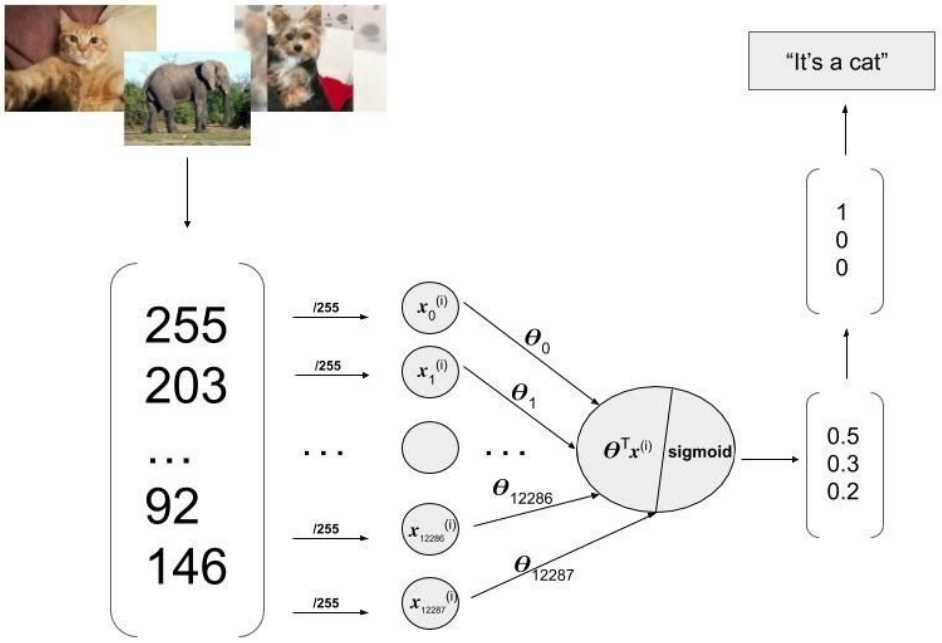
In the first group, we have two implementations.
Input and output in both are similar. For input we have the vector of images, normalized and in shape ((num_px X num_px X 3), m), for our first implementation and (m, (num_px X num_px X 3)) for the second. Where *num_px* is representing the size of the image(64 pixels width and height- we have selected 64 pixels as a balance between losing image quality and increased computational time) and *m*- number of examples. As our inputs are vectors of pixels, their values are integers between 0 and 255. This is the reason why we normalize them dividing by 255.

Our first logistic regression separates the weights and biases and uses a softmax activation function. The image below is the general architecture of it.



In the diagram, the flow of execution can be seen. From the normalized input given to the linear function, later passed through a softmax, to the final output with predictions and One-Hot Encodings.

Our second logistic regression does not separate the weights and biases, it uses thetas(Θ) and a sigmoid activation function. Another difference between this implementation and the one above is that in this, we have included regularization. All of the above-mentioned techniques used in this algorithm have been seen in class and implemented in the practices.
The image below is the general architecture of it.



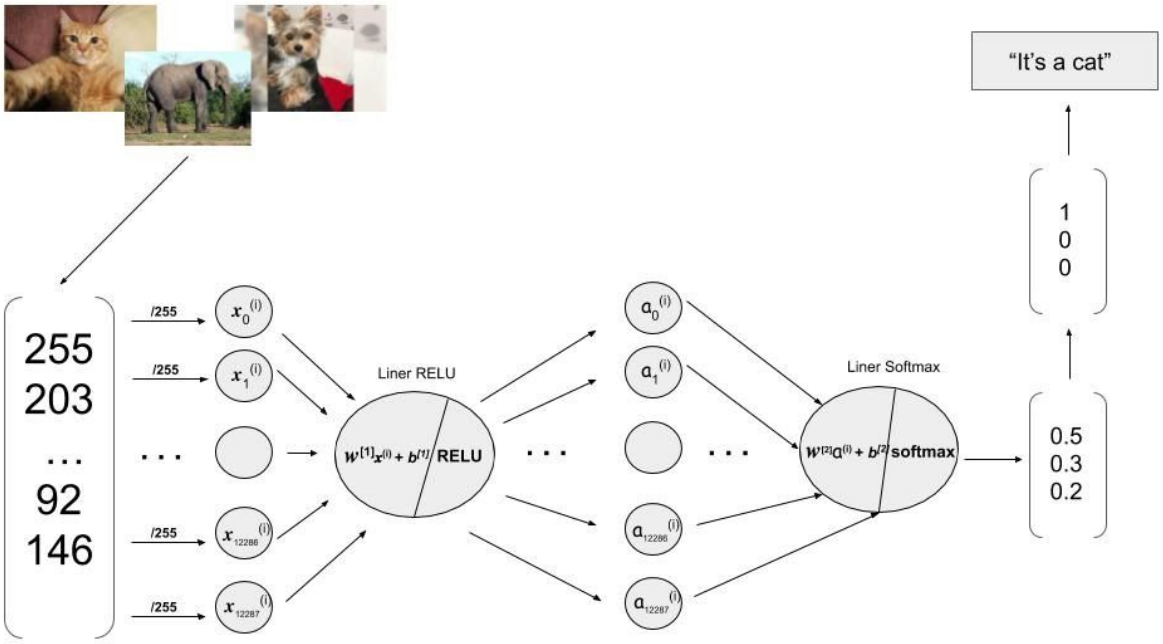
In the diagram, the differences are the thetas(Θ), used in the linear function, and the sigmoid activation function, which replaces the softmax from the first implementation.

Neural network models

In this group, we have two implementations.
Input and output in both are similar to those described for the logistic regression models. The main difference compared to the previous, is that these models use one additional hidden layer, making them more complex but at the same time, capable of distinguishing more advanced features.

Our first neural network implementation is based on the above mentioned, first logistic regression model, where the weights and biases are separated and a softmax activation function is used for the output layer. The hidden layer uses a RELU activation function, chosen by us as it is a very good alternative to sigmoid and broadly used in modern neural network model implementations.

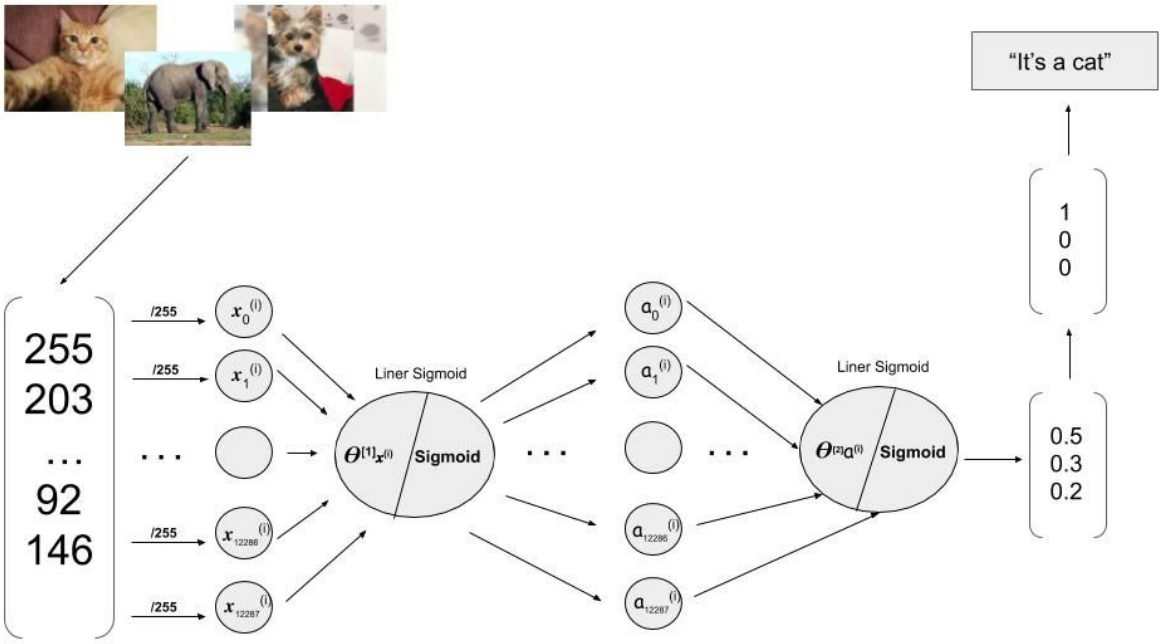
In image below is the general architecture of our first artificial neural network model.



As shown in the above diagram our images, transformed and normalized, pass through a linear function, later through the first activation function(RELU). The result of that is used as an input to the second layer, where the data goes to a linear function and a softmax, generates the final predictions.

The second neural network implementation is base on the above mentioned, second logistic regression model with regularization, where the weights and biases are combined in thetas(Θ), and the activation functions are sigmoids.

In image below is the general architecture of our second artificial neural network model.



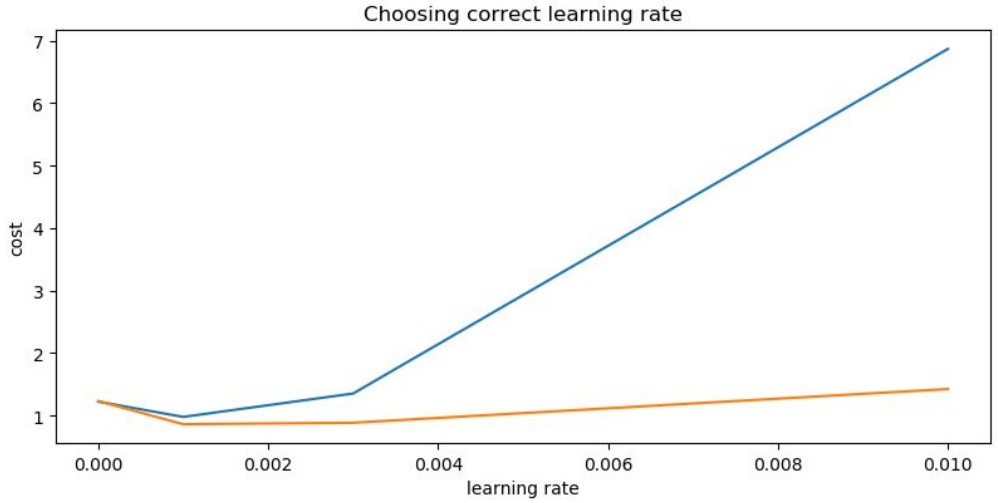
In the diagram, the differences are the thetas(**Θ**), used in the linear function, and the sigmoid activation functions, which replaces the softmax functions from the first implementation.
For more information regarding the code and how to execute it (see Appendix B).

4. Parameters Tuning

As our implementations use different parameters like alpha(**α**), lambda(ƛ), and number of iterations(#iterations). Initially, we were not sure which of them should be tuned or not, and to what extent.

For every model implementation, we created helper functions that run them with different hyperparameter values.

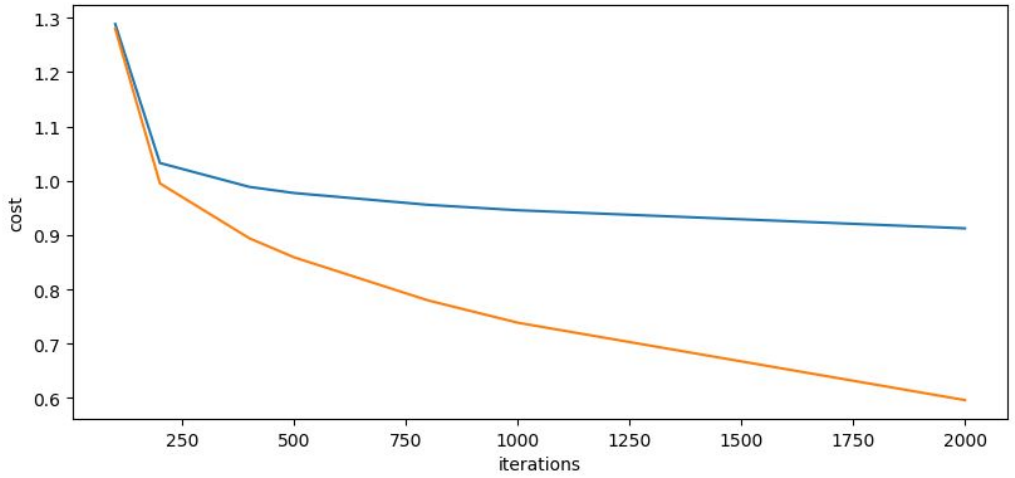
For our logistic regression with softmax, we were tuning the alpha and the number of iterations- as it hasn't got regularization there isn't lambda.
In the graph below is shown the cost of the model, training it with different learning rates: [0, 0.001, 0.003, 0.01]



We can see that in the interval [0.000 and 0.002], concretely 0.001 is the optimal **α**. Another justification is the cost vectors- first(for the blue line, training set) and send(for the orange line, cross-validation set).

```
[1.21997225 0.97543779 1.34992829 6.86854229]
[1.22942828 0.85803612 0.88221132 1.42158336]
```

Next, we search the optimal number of iterations running the models several times with #iterations: [100, 200, 400, 500, 800, 1000, 2000]
The below graph we see that in increasing the number of iterations the cost decreases as expected but after 400, for the training set it nearly goes flat.

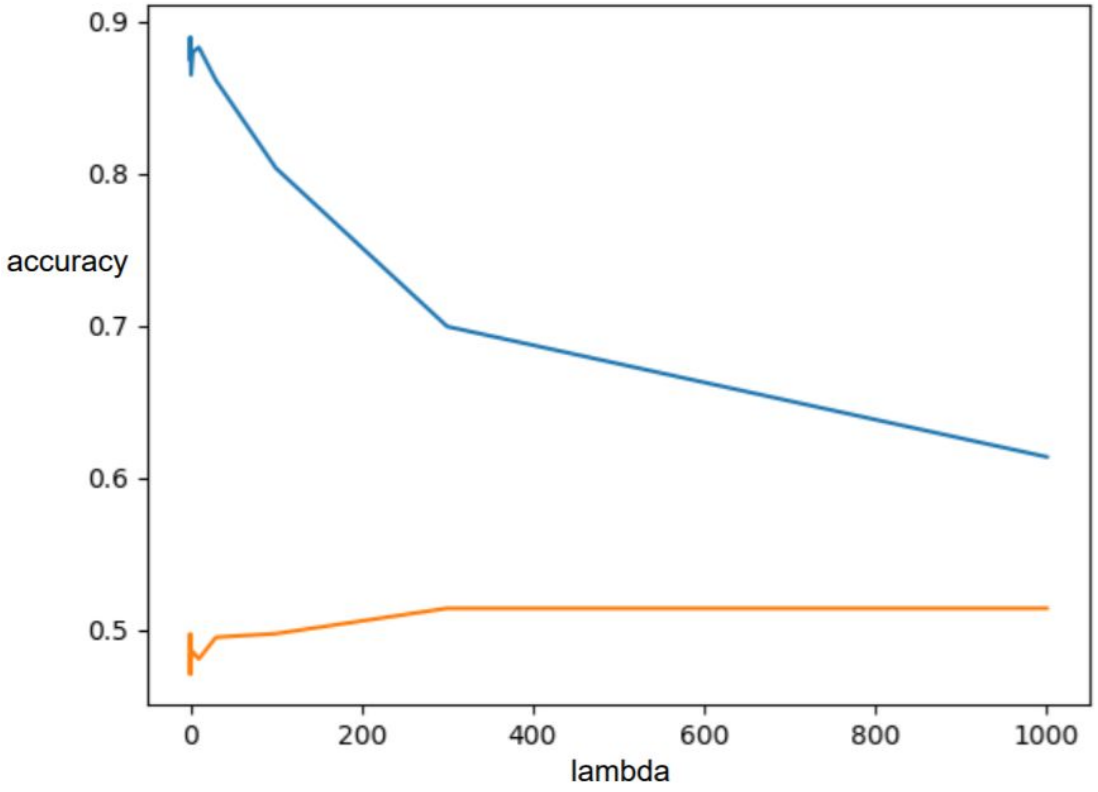


For this algorithm, we will use 400 iterations to calculate the metrics.

For the logistic regression with a sigmoid activation function, our second logistic implementation, we have searched for a lambda(ƛ) that will reduce the high variance between the accuracy of our training set and cross-validation.
In the image below are the accuracies for train set (first list) and cross-validation(second list) obtained running the logistic regression with [0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100, 300, 1000] for ƛ.

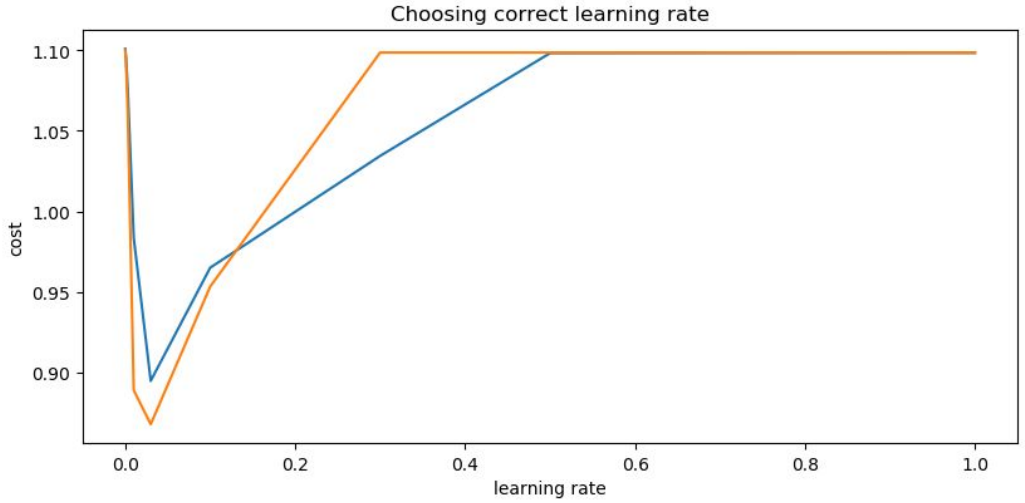
```
[0.88508485 0.87466508 0.8832986  0.88955046 0.87793986 0.88984817
 0.88746651 0.86513843 0.88002382 0.8832986  0.86156594 0.80381066
 0.69961298 0.61387318]
[0.47142857 0.47857143 0.48333333 0.47142857 0.49761905 0.48809524
 0.48571429 0.48095238 0.48571429 0.48095238 0.4952381  0.49761905
 0.51428571 0.51428571]
```

Although the accuracy for the training set is good the variance is high, which pushed us to select a lambda of “300”, so that we can reduce this interval.
The graph above represents the same operation.



For our first neural network model, with RELU and softmax as activation functions, as it is base on the softmax logistic regression we were able to tune only **α** and #iterations.

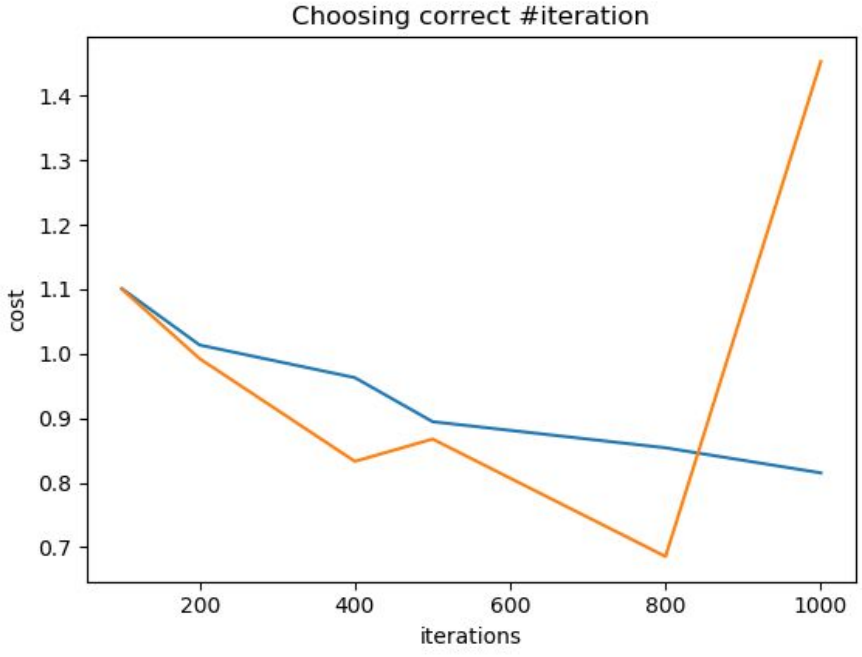
The above graph comes from computing the cost with different learning rates- [0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 0.5, 1].



As shown, the best lambda is between 0.0 and 0.02, more concretely “0.03”. Obtained looking at the costs vectors, where the first is with the results from the train set and the second with the cross-validation one.

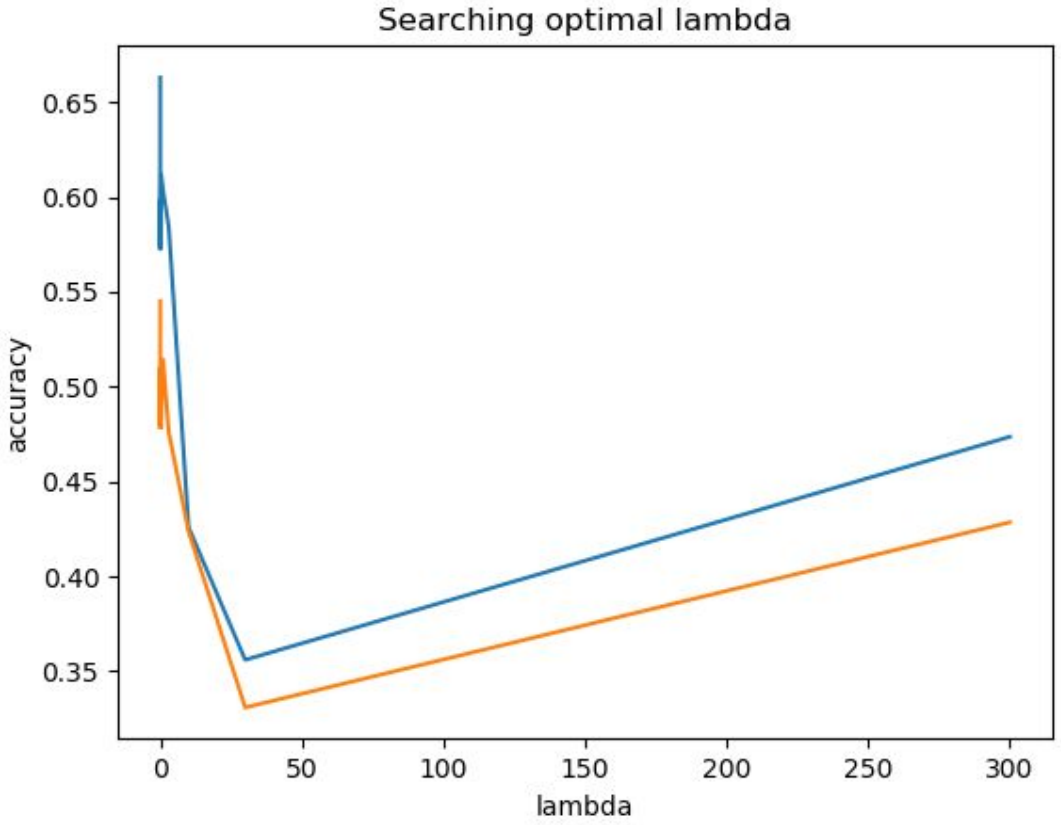
```
[1.10090138 1.09505524 1.07573478 0.98356267 0.89471603 0.96500264
 1.0344228  1.098267   1.0984101 ]
[1.10005003 1.09179583 1.06062589 0.88890834 0.86778805 0.95327444
 1.09861226 1.09861226 1.09861226]
```

Having alpha the next step was calculating the number of iterations. To do so, we executed the model with different #iterations: [100, 200, 400, 500, 800, 1000].
In the graph below are the obtained results.



As shown after 800 iterations the cost of the validation set (the orange line) increases. As a result of that, we decided to use 800 for the number of iterations to limit the gradient explosions.

For the neural network with sigmoids as activation functions, our second neural network implementation, we have searched for a lambda(ƛ) similar to the process we have done for our second logistic regression. We have trained the model with [0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 300] as values for ƛ.
.



Considering the results from the graph, we chose 30 as a value for ƛ in order to obtain better results.

5. Metrics

For all of our implementations, we have calculated the following metrics- precision, recall, average, and F1 Score.

In the following table shows all of the results we have obtained.

					Precision			Recall			Average			F1 Score		
Model		Parameters	Dataset	Accuracy	dogs	cats	elephants	dogs	cats	elephants	dogs	cats	elephants	dogs	cats	elephants
Logistic regression	softmax	iterations = 400	Training	53.14	0.48	0.52	0.58	0.42	0.58	0.59	0.45	0.56	0.58	0.45	0.55	0.58
		α = 0.001	Cross-validation	46.42	0.39	0.44	0.54	0.37	0.42	0.6	0.38	0.43	0.57	0.38	0.43	0.57
			Test	48.57	0.44	0.40	0.63	0.4	0.45	0.60	0.42	0.42	0.62	0.419	0.42	0.62
	sigmoid	iterations = 50	Training	69.96	0.70	0.66	0.73	0.60	0.73	0.76	0.66	0.70	0.70	0.65	0.70	0.75
		ƛ = 300	Cross-validation	51.42	0.44	0.50	0.59	0.39	0.53	0.63	0.41	0.52	0.52	0.41	0.52	0.60
			Test	49.29	0.42	0.42	0.42	0.40	0.46	0.60	0.42	0.44	0.44	0.41	0.44	0.62
Neural Network	RELU/softmax	iterations = 800	Training	46.08	0.38	0.45	0.57	0.40	0.46	0.52	0.39	0.45	0.55	0.39	0.45	0.55
		α = 0.03	Cross-validation	49.76	0.41	0.52	0.57	0.41	0.54	0.54	0.41	0.53	0.55	0.41	0.53	0.55
			Test	84.04	0.79	0.84	0.89	0.76	0.9	0.85	0.77	0.87	0.87	0.77	0.86	0.87
	sigmoids	iterations = 300	Training	75.92	0.75	0.71	0.81	0.63	0.77	0.88	0.69	0.74	0.74	0.69	0.74	0.84
		ƛ = 30	Cross-validation	53.01	0.43	0.54	0.60	0.39	0.54	0.67	0.40	0.54	0.54	0.40	0.54	0.64
			Test	52.62	0.46	0.49	0.62	0.39	0.54	0.42	0.51	0.51	0.42	0.51	0.63	0.63

As expected the neural networks models are presenting better results than the logistic regression ones. If looking at the “F1 Score” for the test sets we can see higher valued camped to the other two. This is a normal behaviour having the fact that these algorithms form deeper networks, fit the data better and distinguish more complex forms.

6. Conclusion

We have implemented four different models in order to resolve a multi-class classification problem- classifying three types of animals.

The models consist of two logistic regressions, softmax and sigmoid, and two neural network-RELU/Softmax and Sigmoids.

The difference in the performance between the two logistic regression implementations is quite insignificant. As both of them are shallow networks and we have possibly reached the limit of their performances.

From the results, we can conclude that the softmax neural network model performs better than the sigmoid one, and respectively is the best of them all.

7. Appendix

Appendix A

In this part of the appendix are described the steps and the code used during the preprocessing of the dataset. All of the files with the code shown below should be created in a directory where the downloaded dataset Animals-10 is located.

Note: Do not change the name of the dataset “59760_840806_bundle_archive.zip” because function “change_folders_names” uses it.

utils.py

```
import numpy as np
import os
from PIL import Image
import matplotlib.pyplot as plt
import os
import imageio
import shutil
```

```
def create_label(Y, folder):
    if folder == "dog":
        val = [1, 0, 0]
        Y.append(val)
    elif folder == "cat":
        val = [0, 1, 0]
        Y.append(val)
    elif folder == "elephant":
        val = [0, 0, 1]
        Y.append(val)


def convert_png_image_to_jpg(path):
    im1 = Image.open(path)
    plt.imshow(im1)

    if im1.mode in ("RGBA", "P"):
        im1 = im1.convert("RGB")

    im1.save(path.replace(".png", ".jpg"))
    if os.path.exists(path):
        os.remove(path)


def covert_all_png_images_to_jpg():
    folder = "images"
    for subfolder in os.listdir(folder):
            subfolder_complete_path = os.path.join(folder, subfolder)
            for filename in os.listdir(subfolder_complete_path):
                if ".png" in filename:
                    print(filename)

convert_png_image_to_jpg(os.path.join(subfolder_complete_path, filename))


def load_images_from_folder(Y, num_px = 64, folder="images"):
    print(f"Loading {folder}.")
    images = []
    for subfolder in os.listdir(folder):
        subfolder_complete_path = os.path.join(folder, subfolder)
```

```
        for filename in os.listdir(subfolder_complete_path):
            img =
np.array(imageio.imread(os.path.join(subfolder_complete_path, filename)))
            if img is not None:
                img = np.array(Image.fromarray(img).resize(size=(num_px,
num_px))) # resize the image
                #print(filename)
                #print(img.shape)
                if img.shape != (num_px, num_px, 3): # delete all images with
icorrect shapes
                    os.remove(os.path.join(subfolder_complete_path, filename))
                    #print(os.path.join(subfolder_complete_path, filename))
                else:
                    img = np.reshape(img, (1, num_px*num_px*3)).T
                    images.append(img)
                    create_label(Y, subfolder)
    images = np.array(images)
    return np.reshape(images, (images.shape[0], images.shape[1])).T


def print_images(images, num_px = 64):
    for index in range(images.shape[1]):
        plt.figure()
        plt.imshow(images[:,index].reshape((num_px, num_px, 3)))


def separate_dataset(X):
    print("a")
    print(X.shape)
    np.random.shuffle(X.T)
    print(X)
    print(np.random.shuffle(X.T))
    print(X.shape)

    #print_images(X)

    X_splitted = X[:]
    X_train = X[:,:5]
    X_dev = X_splitted[:,5:8]
    X_test = X_splitted[:,8:9]
```

```
    print("shapes")
    print(X_train.shape)
    print(X_dev.shape)
    print(X_test.shape)

    print_images(X_train)
    #print_images(X_dev)
    #print_images(X_test)

    plt.show()


def separate_dataset_folders(max_examples = 1400):
    # max_examples is the number of files found for every
    # class(in this case as for class-elephant, there are only 1440
    # although for dog and cats there are more this the maximum number
    # of images that our model is going to use)

    train = max_examples * 80 / 100
    dev = max_examples * 10 / 100
    test = max_examples * 10 / 100

    folder = "images"
    folder_train = "images_train"
    folder_dev = "images_dev"
    folder_test = "images_test"

    for subfolder in os.listdir(folder):
            subfolder_complete_path = os.path.join(folder, subfolder)
            subfolder_complete_path_train = os.path.join(folder_train,
subfolder)
            subfolder_complete_path_dev = os.path.join(folder_dev, subfolder)
            subfolder_complete_path_test = os.path.join(folder_test,
subfolder)

            for i in [subfolder_complete_path_train,
subfolder_complete_path_dev, subfolder_complete_path_test]:
                if not os.path.exists(i):
                    os.makedirs(i)
```

```
            count_train = 0
            count_dev = 0
            count_test = 0
            files_read = 0

            for filename in os.listdir(subfolder_complete_path):
                print(filename)
                if max_examples > files_read:
                    if count_train < train:
                        os.replace(os.path.join(subfolder_complete_path,
filename), os.path.join(subfolder_complete_path_train, filename))
                        count_train += 1
                    elif count_dev < dev:
                        os.replace(os.path.join(subfolder_complete_path,
filename), os.path.join(subfolder_complete_path_dev, filename))
                        count_dev += 1
                    elif count_test < test:
                        os.replace(os.path.join(subfolder_complete_path,
filename), os.path.join(subfolder_complete_path_test, filename))
                        count_test += 1
                    files_read += 1
                else:
                    break

    shutil.rmtree(folder)


def delete_not_translated_folders(dog_folder, cat_folder, elephant_folder,
images_path):
    for directory in os.listdir(images_path):
                directory = os.path.join(images_path, directory)
                if directory not in [dog_folder, elephant_folder, cat_folder]:
                     shutil.rmtree(directory)


def change_folders_names():
    folder_name = "59760_840806_bundle_archive"
    import zipfile

    with zipfile.ZipFile(folder_name + ".zip", 'r') as zip_ref:
```

```
        zip_ref.extractall(folder_name)


    dog_folder = "cane"
    elephant_folder = "elefante"
    cat_folder = "gatto"
    if os.path.exists(folder_name):
        images_path = os.path.join(folder_name, "raw-img")
        if os.path.exists(images_path):
            if os.path.exists("images"):
                os.rmdir("images")

            os.rename(images_path, "images")
            images_path = os.path.join("images", "")

            if os.path.exists(folder_name):
                shutil.rmtree(folder_name)

            dog_folder = os.path.join(images_path, dog_folder)

            if os.path.exists(dog_folder):
                 os.rename(dog_folder, os.path.join(images_path, "dog"))
                 dog_folder = os.path.join(images_path, "dog")

            elephant_folder = os.path.join(images_path, elephant_folder)

            if os.path.exists(elephant_folder):
                 os.rename(elephant_folder, os.path.join(images_path,
"elephant"))
                 elephant_folder = os.path.join(images_path, "elephant")

            cat_folder = os.path.join(images_path, cat_folder)

            if os.path.exists(cat_folder):
                 os.rename(cat_folder, os.path.join(images_path, "cat"))
                 cat_folder = os.path.join(images_path, "cat")

            delete_not_translated_folders(dog_folder, cat_folder,
elephant_folder, images_path)
```

```
def prepare_dataset():
    change_folders_names()
    covert_all_png_images_to_jpg()
    separate_dataset_folders()
```

clean_dataset.py

```
from utils import prepare_dataset

prepare_dataset()
```

In a terminal with python execute the following command.
Note: During development, we used Visual Studio Code from Anaconda and its terminal. This way the scripts run directly without the need of any other dependencies.
If they will be executed differently - other environments, they may need a previous installation of all of the dependencies imported into the file "utils.py".

```
python clean_dataset.py
```

Appendix B

In this part of the appendix are described the code and the steps to execute it. The code represents our four different implementations described in Models, their parameters tuning, and their Metrics.

Note: All of the python scripts should be created in the same directory where the steps from Appendix A were executed.

logistic_regression_softmax.py

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
from utils import load_images_from_folder, print_images

Y_train = [] # labels are created during execution time
Y_dev = []
```

```
Y_test = []
num_px = 64 # during execution images are resized to 64x64x3. This way we lose
their quality but we save computational time.
C = 3 # number of classes to detect

folder_train = "images_train"
folder_dev = "images_dev"
folder_test = "images_test"

images_train = load_images_from_folder(Y_train, folder= folder_train)
X_train = images_train / 255 # normalize dataset

images_dev = load_images_from_folder(Y_dev, folder= folder_dev)
X_dev = images_dev / 255 # normalize dataset

images_test = load_images_from_folder(Y_test, folder= folder_test)
X_test = images_test / 255 # normalize dataset

Y_train = np.array(Y_train) # convert the labels Y list into a numpy array
Y_dev = np.array(Y_dev)
Y_test = np.array(Y_test)

### activation function - softmax
def softmax(z):
    t = np.exp(z)
    return t / np.sum(t, axis= 0)


def initialize(dim):

    w = np.random.randn(dim, C) * 0.01
    b = 0

    return w, b


def propagate(w, b, X, Y):
    m = X.shape[1]

    A = softmax(np.dot(w.T, X) + b)
```

```
    cost = np.sum(Y * np.log(A.T) + 1e-8) / -m
    dw = np.dot(X, np.transpose(A - Y.T)) / m
    db = np.sum((A - Y.T)) / m

    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())

    grads = {"dw": dw,
             "db": db}

    return grads, cost


w, b = initialize(X_train.shape[0])
grads, cost = propagate(w, b, X_train, Y_train)


def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False): #
clean the code
    costs = []

    print(f"Learning rate {learning_rate}")

    for i in range(num_iterations):
        # Compute gradients and cost
        grads, cost = propagate(w, b, X, Y)

        # Gradients
        dw = grads["dw"]
        db = grads["db"]

        # Update weight and bias
        w = w - learning_rate * dw
        b = b - learning_rate * db

        # Record the costs
        if i % 100 == 0:
            costs.append(cost)
```

```
        # Print the cost every 100 training iterations
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, costs

def one_hot_reverse(Y):
    H = np.argmax(Y, axis=1)
    return H.reshape((len(H), 1))

def predict(w, b, X, Y):
    A = softmax(np.dot(w.T, X) + b)

    index_max = np.argmax(A, axis = 0)
    index_max = index_max.reshape((len(index_max), 1))
    accuracy = np.sum(one_hot_reverse(Y) % C == index_max) / Y.shape[0]
    print("Accuracy: " + str(accuracy * 100) + '%')

def predict_one_example(index):
    softmax_picture = softmax(np.dot(X[:,index], params["w"]) + params["b"])
    type_animal = np.argmax(softmax_picture)
    message = ""
    if type_animal == 0:
        message += "I am a dog "
    elif type_animal == 1:
        message += "I am a cat"
    elif type_animal == 2:
        message += "I am an elephant"
    message += "with probability of " +
"{:.2f}".format(softmax_picture[type_animal] * 100) + "%."
    plt.figure()
    plt.xlabel(message)
    plt.imshow(images[:,index].reshape((num_px, num_px, 3)))
```

```
#predict_one_example(2001)

plt.show()

# Precision/Recall


def choose_rate(X, y, X_val, y_val):
    lr = np.array([0, 0.001, 0.003, 0.01]) #0.03, 0.1, 0.3
    costs_X = np.zeros(len(lr))
    costs_X_val = np.zeros(len(lr))
    for rate in range(len(lr)):
            _, _, cost_X = optimize(w, b, X, y, num_iterations= 500,
learning_rate = lr[rate], print_cost = True)
            costs_X[rate] = cost_X[-1]
            _, _, cost_X_val = optimize(w, b, X_val, y_val, num_iterations=
500, learning_rate = lr[rate], print_cost = True)
            costs_X_val[rate] = cost_X_val[-1]
    print(costs_X)
    print(costs_X_val)
    plt.plot(lr, costs_X)
    plt.xlabel("learning rate")
    plt.plot(lr, costs_X_val)
    plt.ylabel("cost")
    plt.title("Choosing correct learning rate")
    plt.show()

#choose_rate(X_train, Y_train, X_dev, Y_dev)


def choose_iterations(X, y, X_val, y_val):
    lr = np.array([100, 200, 400, 500, 800, 1000, 2000])
    costs_X = np.zeros(len(lr))
    costs_X_val = np.zeros(len(lr))
    for rate in range(len(lr)):
            _, _, cost_X = optimize(w, b, X, y, num_iterations= lr[rate],
learning_rate = 0.001, print_cost = True)
            costs_X[rate] = cost_X[-1]
            _, _, cost_X_val = optimize(w, b, X_val, y_val, num_iterations=
```

```
lr[rate], learning_rate = 0.001, print_cost = True)
            costs_X_val[rate] = cost_X_val[-1]
    print(costs_X)
    print(costs_X_val)
    plt.plot(lr, costs_X)
    plt.xlabel("iterations")
    plt.plot(lr, costs_X_val)
    plt.ylabel("cost")
    plt.show()

#choose_iterations(X_train, Y_train, X_dev, Y_dev)

def calculate_probability(w, b, X, Y, C=3):
    A = softmax(np.dot(w.T, X) + b)

    index_max = np.argmax(A, axis = 0)
    index_max = index_max.reshape((len(index_max), 1))
    accuracy = np.sum(one_hot_reverse(Y) % C == index_max) / Y.shape[0]
    print("Accuracy: " + str(accuracy * 100) + '%')

    precision = np.zeros(C)
    recall = np.zeros(C)
    for c in range(C):
        precision[c] = np.sum((Y[:, c] == 1) * (index_max == c).ravel()) /
np.sum(index_max == c)
        recall[c] = np.sum((Y[:, c] == 1) * (index_max == c).ravel()) /
np.sum(Y[:, c] == 1)
    print("precision dogs: " + str(precision[0]))
    print("precision cats: " + str(precision[1]))
    print("precision elephants: " + str(precision[2]))

    print("recall dogs: " + str(recall[0]))
    print("recall cats: " + str(recall[1]))
    print("recall elephants: " + str(recall[2]))

    print("average dogs: " + str((precision[0] + recall[0]) / 2))
    print("average cats: " + str((precision[1] + recall[1]) / 2))
    print("average elephants: " + str((precision[2] + recall[2]) / 2))

    f1_score_dogs = 2 * precision[0] * recall[0] / (precision[0] + recall[0])
```

```
    print("f1 score dogs: " + str(f1_score_dogs))
    f1_score_cats = 2 * precision[1] * recall[1] / (precision[1] + recall[1])
    print("f1 score cats: " + str(f1_score_cats))
    f1_score_elephants = 2 * precision[2] * recall[2] / (precision[2] +
recall[2])
    print("f1 score elephants: " + str(f1_score_elephants))

# Running the model
params, grads, costs = optimize(w, b, X_train, Y_train, num_iterations= 400,
learning_rate = 0.001, print_cost = True)

print("Accuracy training set:")
calculate_probability(params["w"], params["b"], X_train, Y_train)
print("Accuracy validation set:")
calculate_probability(params["w"], params["b"], X_dev, Y_dev)
print("Accuracy test set:")
calculate_probability(params["w"], params["b"], X_test, Y_test)
```

Run the model executing:

```
python logistic_regression_softmax.py
```

logistic_regression_sigmoid.py

```
import numpy as np
import matplotlib.pyplot as plt
import scipy
from scipy import optimize as opt
from scipy import ndimage
from utils import load_images_from_folder
from sklearn.preprocessing import PolynomialFeatures

Y_train = [] # labels are created during execution time
Y_dev = []
Y_test = []
num_px = 64 # during execution images are resized to 64x64x3. This way we lose
their quality but we save computational time.
C = 3 # number of classes to detect

images_train = load_images_from_folder(Y_train, folder="images_train")
X_train = images_train / 255 # normalize dataset
```

```
Y_train = np.array(Y_train) # convert the labels Y list into a numpy array

images_dev = load_images_from_folder(Y_dev, folder="images_dev")
X_dev = images_dev / 255 # normalize dataset
Y_dev = np.array(Y_dev) # convert the labels Y list into a numpy array

images_test = load_images_from_folder(Y_test, folder="images_test")
X_test = images_test / 255 # normalize dataset
Y_test = np.array(Y_test) # convert the labels Y list into a numpy array

X_train = X_train.T
X_train = np.hstack([np.ones([len(X_train), 1]), X_train])
X_dev = X_dev.T
X_dev = np.hstack([np.ones([len(X_dev), 1]), X_dev])
X_test = X_test.T
X_test = np.hstack([np.ones([len(X_test), 1]), X_test])

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

def cost(theta, X, Y, lambd):
    theta = theta.reshape((len(theta), 1))
    A = sigmoid(np.matmul(X, theta))
    reg = (lambd / (2 * len(X))) * np.sum(theta ** 2)
    return (- 1 / (len(X))) * np.sum(Y * np.log(A) + (1 - Y) * np.log(1 - A  +
1e-6)) + reg

def gradient(theta, X, Y, lambd):
    theta = theta.reshape((len(theta), 1))
    A = sigmoid(np.matmul(X, theta))
    identity = np.identity(theta.shape[0])
    identity[0][0] = 0
    return (np.dot(X.T, (A - Y)) / len(Y) + (lambd / len(X)) *
np.dot(identity, theta)).ravel()

def model(X, Y, lambd):
    theta = np.zeros(X.shape[1])
    result = opt.fmin_tnc(func=cost, x0=theta, fprime=gradient, args = (X, Y,
lambd), maxfun=50)
    theta = result[0]
```

```
    return theta

def oneVsAll(X, Y, n_labels, reg):
    """
    oneVsAll entrena varios clasificadores por regresión logística con término
    de regularización 'reg' y devuelve el resultado en una matriz, donde
    la fila i-ésima corresponde al clasificador de la etiqueta i-ésima
    """
    theta = []
    for i in range(n_labels):
        theta.append(model(X, Y[:, i].reshape((len(Y), 1)), reg))
    theta = np.array(theta)
    return theta

def choose_lambda(X, y, X_val, y_val, C):
    lambd = np.array([0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30,
100])
    error_dots = np.zeros(len(lambd))
    error_val = np.zeros(len(lambd))
    for i in range(len(lambd)):
        theta = oneVsAll(X, y, C, lambd[i])
        for c in range(C):
            error_dots[i] += gradient(theta[c, :], X, y, lambd[i])[0]
            error_val[i] += gradient(theta[c, :], X_val, y_val, lambd[i])[0]
        error_dots[i] /= C
        error_val[i] /= C
    plt.plot(lambd, error_dots)
    plt.plot(lambd, error_val)
    print(error_dots)
    print(error_val)
    plt.show()

def choose_lambda_acc(X, Y, X_val, Y_val, C=3):
    lambd = np.array([0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30,
100, 300, 1000])
    accuracy_dots = np.zeros(len(lambd))
    accuracy_val = np.zeros(len(lambd))

    for i in range(len(lambd)):
        theta = oneVsAll(X, Y, C, lambd[i])
```

```
        prediction = np.dot(X, theta.T)
        index_max = np.argmax(prediction, axis = 1)
        index_max = index_max.reshape((len(index_max), 1))
        accuracy_dots[i] = np.sum(np.argmax(Y, axis = 1).reshape((len(Y), 1))
== index_max) / Y.shape[0]

        predict_val = np.dot(X_val, theta.T)
        index_max = np.argmax(predict_val, axis = 1)
        index_max = index_max.reshape((len(index_max), 1))
        accuracy_val[i] = np.sum(np.argmax(Y_val, axis =
1).reshape((len(Y_val), 1)) == index_max) / Y_val.shape[0]
    plt.plot(lambd, accuracy_dots)
    plt.plot(lambd, accuracy_val)
    print(accuracy_dots)
    print(accuracy_val)
    plt.show()

def calculate_probability(X, Y, theta, C=3):
    prediction = np.dot(X, theta.T)
    index_max = np.argmax(prediction, axis = 1)
    index_max = index_max.reshape((len(index_max), 1))
    accuracy = np.sum(np.argmax(Y, axis = 1).reshape((len(Y), 1)) ==
index_max) / Y.shape[0]
    print("Accuracy " + str(accuracy * 100) + '%')

    precision = np.zeros(C)
    recall = np.zeros(C)
    for c in range(C):
        precision[c] = np.sum((Y[:, c] == 1) * (index_max == c).ravel()) /
np.sum(index_max == c)
        recall[c] = np.sum((Y[:, c] == 1) * (index_max == c).ravel()) /
np.sum(Y[:, c] == 1)
    print("precision dogs: " + str(precision[0]))
    print("precision cats: " + str(precision[1]))
    print("precision elephants: " + str(precision[2]))

    print("recall dogs: " + str(recall[0]))
    print("recall cats: " + str(recall[1]))
    print("recall elephants: " + str(recall[2]))
```

```
    print("f1 score dogs: " + str(2 * precision[0] * recall[0] / (precision[0]
+ recall[0])))
    print("f1 score cats: " + str(2 * precision[1] * recall[1] / (precision[1]
+ recall[1])))
    print("f1 score elephants: " + str(2 * precision[2] * recall[2] /
(precision[2] + recall[2])))

    print("average dogs: " + str((precision[0] + recall[0]) / 2))
    print("average cats: " + str((precision[1] + recall[1]) / 2))
    print("average elephants: " + str((precision[1] + recall[1]) / 2))




#choose_lambda(X_train, Y_train, X_dev, Y_dev, 3)
#choose_lambda_acc(X_train, Y_train, X_dev, Y_dev, 3)
thetas = oneVsAll(X_train, Y_train, C, 300)
print("Accuracy training set:")
calculate_probability(X_train, Y_train, thetas)
print("Accuracy validation set:")
calculate_probability(X_dev, Y_dev, thetas)
print("Accuracy test set:")
calculate_probability(X_test, Y_test, thetas)

#theta = np.zeros(X.shape[1])
#print(theta.shape)
#print(gradient(theta, X, Y[:, 0].reshape((len(Y), 1)), 0.1).shape)
#print(cost(theta, X, Y, 0.1))
```

Run the model executing:

```
python logistic_regression_sigmoid.py
```

neural_network_softmax.py

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
from utils import load_images_from_folder, print_images
```

```
Y_train = [] # labels are created during execution time
Y_dev = []
Y_test = []
num_px = 64 # during execution images are resized to 64x64x3. This way we lose
their quality but we save computational time.
C = 3 # number of classes to detect

folder_train = "images_train"
folder_dev = "images_dev"
folder_test = "images_test"

images_train = load_images_from_folder(Y_train, folder= folder_train)
X_train = images_train / 255 # normalize dataset

images_dev = load_images_from_folder(Y_dev, folder= folder_dev)
X_dev = images_dev / 255 # normalize dataset

images_test = load_images_from_folder(Y_test, folder= folder_test)
X_test = images_test / 255 # normalize dataset

Y_train = np.array(Y_train) # convert the labels Y list into a numpy array
Y_dev = np.array(Y_dev)
Y_test = np.array(Y_test)


def softmax(z):
    t = np.exp(z)
    A = t / np.sum(t, axis= 0)
    cache = z
    return A, cache

def relu(Z):
    A = np.maximum(0,Z)

    assert(A.shape == Z.shape)

    cache = Z
    return A, cache
```

```
def relu_backward(dA, cache):

    Z = cache
    dZ = np.array(dA, copy=True)

    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def initialize_parameters(n_x, n_h, n_y):
    np.random.seed(1)

    W1 = np.random.randn(n_h, n_x)*0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h)*0.01
    b2 = np.zeros((n_y, 1))

    assert(W1.shape == (n_h, n_x))
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters

def initialize_parameters_deep(layer_dims):
    np.random.seed(3)
    parameters = {}
    L = len(layer_dims)

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l
- 1]) * 0.01
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))
```

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


    return parameters

def linear_forward(A, W, b):
    Z = np.dot(W, A) + b

    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache

def linear_activation_forward(A_prev, W, b, activation):
    if activation == "relu":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)
    elif activation == "softmax":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = softmax(Z)
    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache

def L_model_forward(X, parameters, C):
    caches = []
    A = X
    L = len(parameters) // 2  # number of layers in the neural network

    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev,parameters["W" + str(l)],
parameters["b" + str(l)], "relu")
        caches.append(cache)
    AL, cache = linear_activation_forward(A,parameters["W" + str(L)],
parameters["b" + str(L)], "softmax")
```

```
    caches.append(cache)
    assert(AL.shape == (C, X.shape[1]))

    return AL, caches

def linear_backward(dZ, cache):
    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = np.dot(dZ, A_prev.T) / m
    db = np.sum(dZ, axis=1, keepdims = True) / m
    dA_prev = np.dot(W.T, dZ)

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db

def linear_activation_backward(dA, cache, activation):
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA, cache[1])
        dA_prev, dW, db = linear_backward(dZ, cache[0])
    if activation == "softmax":
        dZ = dA
        dA_prev, dW, db = linear_backward(dZ, cache[0])

    return dA_prev, dW, db

def L_model_backward(AL, Y, caches):
    grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    #Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

    dAL = AL - Y.T

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "dAL, current_cache".
```

```
Outputs: "grads["dAL-1"], grads["dWL"], grads["dbL"]
    current_cache = linear_activation_backward(dAL, caches[L - 1], "softmax")
    grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" + str(L)] =
current_cache

    # Loop from l=L-2 to l=0
    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        # Inputs: "grads["dA" + str(l + 1)], current_cache". Outputs:
"grads["dA" + str(l)] , grads["dW" + str(l + 1)] , grads["db" + str(l + 1)]
        current_cache = linear_activation_backward(grads["dA" + str(l + 1)],
caches[l], "relu")
        dA_prev_temp, dW_temp, db_temp = current_cache
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads

def update_parameters(parameters, grads, learning_rate):
    L = len(parameters) // 2 # number of layers in the neural network

    for l in range(L):
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
learning_rate * grads["dW" + str(l+1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
learning_rate * grads["db" + str(l+1)]

    return parameters

def compute_cost(X, AL, Y):
    m = X.shape[1]
    cost = np.sum(Y * np.log(AL.T) + 1e-8) / -m
    return cost

def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations =
3000, print_cost=False, C = 3):
    np.random.seed(1)
    costs = []                         # keep track of cost
```

```
    parameters = initialize_parameters_deep(layers_dims)

    # Loop (gradient descent)
    for i in range(0, num_iterations):
        AL, caches = L_model_forward(X, parameters, C)

        cost = compute_cost(X, AL, Y)
        grads = L_model_backward(AL, Y, caches)

        parameters = update_parameters(parameters, grads, learning_rate)

        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))
        if print_cost and i % 100 == 0:
            costs.append(cost)

    # plot the cost
    '''
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per hundreds)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()
    '''
    return parameters, costs

def one_hot_reverse(Y):
    H = np.argmax(Y, axis=1)
    return H.reshape((len(H), 1))

def predict_nn(X, Y, parameters, C=3):
    probas, caches = L_model_forward(X, parameters, C)

    index_max = np.argmax(probas, axis = 0)
    index_max = index_max.reshape((len(index_max), 1))
    accuracy = np.sum(one_hot_reverse(Y) % C == index_max) / Y.shape[0]
    print("Accuracy: " + str(accuracy * 100) + '%')

def predict_one_example_nn(index, X, parameters, C=3):
    probas, caches = L_model_forward(X, parameters, C)
```

```
    type_animal = np.argmax(probas, axis = 0)
    if type_animal[index] == 0:
        print("I am a dog", end =" ")
    elif type_animal[index] == 1:
        print("I am a cat", end =" ")
    elif type_animal[index] == 2:
        print("I am an elephant", end =" ")
    probabilty = probas[:,index][np.argmax(probas[:,index])]
    print("with probability of " + "{:.2f}".format(probabilty * 100) + "%.")
    plt.figure()
    #plt.imshow(images[:,index].reshape((num_px, num_px, 3)))

#predict_one_example_nn(2291, X, parameters)


C = 3
#layers_dims = [12288, 20, 7, 5, C] #  4-layer model
#layers_dims = [12288, 20, C] #  2-layer model learning_rate = 0.005, 10000
iterations accuracy = 79.5
layers_dims = [12288, 40, C] # 2 layers with learning_rate = 0.0075,
num_iterations = 20000 accuracy=100
#layers_dims = [12288, 20, 7, C] #  3-layer model learning_rate = 0.009, 10000
iterations accuracy = 68
#params, _ = L_layer_model(X_test, Y_test, layers_dims, learning_rate =
0.0075, num_iterations = 100, print_cost = True, C = C)


#predict_nn(X_test, Y_test, params)

def calculate_probability(parameters, X, Y, C=3):
    probas, _ = L_model_forward(X, parameters, C)
    index_max = np.argmax(probas, axis = 0)
    index_max = index_max.reshape((len(index_max), 1))
    accuracy = np.sum(one_hot_reverse(Y) % C == index_max) / Y.shape[0]
    print("Accuracy: " + str(accuracy * 100) + '%')

    precision = np.zeros(C)
    recall = np.zeros(C)
    for c in range(C):
        precision[c] = np.sum((Y[:, c] == 1) * (index_max == c).ravel()) /
```

```
np.sum(index_max == c)
        recall[c] = np.sum((Y[:, c] == 1) * (index_max == c).ravel()) /
np.sum(Y[:, c] == 1)
    print("precision dogs: " + str(precision[0]))
    print("precision cats: " + str(precision[1]))
    print("precision elephants: " + str(precision[2]))

    print("recall dogs: " + str(recall[0]))
    print("recall cats: " + str(recall[1]))
    print("recall elephants: " + str(recall[2]))

    print("average dogs: " + str((precision[0] + recall[0]) / 2))
    print("average cats: " + str((precision[1] + recall[1]) / 2))
    print("average elephants: " + str((precision[2] + recall[2]) / 2))

    f1_score_dogs = 2 * precision[0] * recall[0] / (precision[0] + recall[0])
    print("f1 score dogs: " + str(f1_score_dogs))
    f1_score_cats = 2 * precision[1] * recall[1] / (precision[1] + recall[1])
    print("f1 score cats: " + str(f1_score_cats))
    f1_score_elephants = 2 * precision[2] * recall[2] / (precision[2] +
recall[2])
    print("f1 score elephants: " + str(f1_score_elephants))

def choose_rate(X, y, X_val, y_val):
    lr = np.array([0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 0.5, 1])
    costs_X = np.zeros(len(lr))
    costs_X_val = np.zeros(len(lr))
    for rate in range(len(lr)):
            _ , cost_X = L_layer_model(X, y, layers_dims, learning_rate =
lr[rate], num_iterations = 500, print_cost = True)
            costs_X[rate] = cost_X[-1]
            _, cost_X_val = L_layer_model(X_val, y_val, layers_dims,
learning_rate = lr[rate], num_iterations = 500, print_cost = True)
            costs_X_val[rate] = cost_X_val[-1]
    print(costs_X)
    print(costs_X_val)
    plt.plot(lr, costs_X)
    plt.xlabel("learning rate")
    plt.plot(lr, costs_X_val)
    plt.ylabel("cost")
```

```
    plt.title("Choosing correct learning rate")
    plt.show()

#choose_rate(X_train, Y_train, X_dev, Y_dev)

def choose_iterations(X, y, X_val, y_val):
    lr = np.array([100, 200, 400, 500, 800, 1000])
    costs_X = np.zeros(len(lr))
    costs_X_val = np.zeros(len(lr))
    for rate in range(len(lr)):
            _ , cost_X = L_layer_model(X, y, layers_dims, learning_rate =
0.03, num_iterations = lr[rate], print_cost = True)
            costs_X[rate] = cost_X[-1]
            _, cost_X_val = L_layer_model(X_val, y_val, layers_dims,
learning_rate = 0.03, num_iterations = lr[rate], print_cost = True)
            costs_X_val[rate] = cost_X_val[-1]
    print(costs_X)
    print(costs_X_val)
    plt.plot(lr, costs_X)
    plt.xlabel("iterations")
    plt.plot(lr, costs_X_val)
    plt.ylabel("cost")
    plt.title("Choosing correct #iteration")
    plt.show()

#choose_iterations(X_train, Y_train, X_dev, Y_dev)

params, _ = L_layer_model(X_test, Y_test, layers_dims, learning_rate = 0.03,
num_iterations = 800, print_cost = True, C = C)

print("Accuracy training set:")
calculate_probability(params, X_train, Y_train)
print("Accuracy validation set:")
calculate_probability(params, X_dev, Y_dev)
print("Accuracy test set:")
calculate_probability(params, X_test, Y_test)
```

Run the model executing:

```
python neural_network_softmax.py
```

This model uses 40 neurons in the hidden layer.

neural_network_sigmoid.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize as opt
from scipy.io import loadmat
from sklearn.preprocessing import PolynomialFeatures
import scipy.optimize as opt
from utils import load_images_from_folder
from sklearn.preprocessing import PolynomialFeatures

Y_train = [] # labels are created during execution time
Y_dev = []
Y_test = []
num_px = 64 # during execution images are resized to 64x64x3. This way we lose
their quality but we save computational time.
C = 3 # number of classes to detect

images_train = load_images_from_folder(Y_train, folder="images_train")
X_train = images_train / 255 # normalize dataset
Y_train = np.array(Y_train) # convert the labels Y list into a numpy array

images_dev = load_images_from_folder(Y_dev, folder="images_dev")
X_dev = images_dev / 255 # normalize dataset
Y_dev = np.array(Y_dev) # convert the labels Y list into a numpy array

images_test = load_images_from_folder(Y_test, folder="images_test")
X_test = images_test / 255 # normalize dataset
Y_test = np.array(Y_test) # convert the labels Y list into a numpy array

X_train = X_train.T
X_train = np.hstack([np.ones([len(X_train), 1]), X_train])
X_dev = X_dev.T
X_dev = np.hstack([np.ones([len(X_dev), 1]), X_dev])
X_test = X_test.T
X_test = np.hstack([np.ones([len(X_test), 1]), X_test])

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

```
def derivade_sigmoid(dA):
    return dA * (1 - dA)

def pesosAleatorios(L_in, L_out, epsilon = 0.12):
    #devolverá una matriz de dimensión (L_out, 1 + L_in)
    return np.random.rand(L_out, 1 + L_in) * (epsilon + epsilon) - epsilon

def linear_activation_forward(A_prev, theta):
    Z = np.dot(A_prev, theta.T)
    A = sigmoid(Z)
    return A

def L_model_forward(X, parameters):
    A = X
    cache = {}
    L = len(parameters)
    for l in range(L):
        A_prev = A
        A_prev = np.hstack([np.ones([A_prev.shape[0], 1]), A_prev])
        cache["A" + str(l + 1)] = A_prev
        A = linear_activation_forward(A_prev, parameters['theta' + str(l +
1)])
    cache["A" + str(L + 1)] = A
    return (A, cache)

def cost(parameters, A, Y, lambd, m):
    reg = (lambd / (2 * m)) * (np.sum(parameters["theta1"][:, 1:] ** 2) +
np.sum(parameters["theta2"][:, 1:] ** 2))
    coste = (Y * np.log(A)) + ((1 - Y) * np.log( 1 - A) )
    return (- 1 / m) * coste.sum() + reg

def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, reg):
    parameters = {}
    grads = {}
    grads["dT1"] = 0
    grads["dT2"] = 0
    params_rn = params_rn.reshape(len(params_rn), 1)
    theta1 = np.reshape(params_rn[: num_ocultas * (num_entradas + 1)],
(num_ocultas, (num_entradas + 1)))
```

```
    theta2 = np.reshape(params_rn[num_ocultas * (num_entradas + 1) :],
(num_etiquetas, (num_ocultas + 1)))

    parameters['theta1'] = theta1
    parameters['theta2'] = theta2
    AL, cache = L_model_forward(X, parameters)
    coste = cost(parameters, AL, y, reg, len(X))
    grads["dA3"] = AL - y
    grads["dA2"] = np.dot(grads["dA3"], parameters['theta2']) *
derivade_sigmoid(cache["A2"])

    grads["dT1"] += (np.dot(grads["dA2"][:, 1:].T, cache["A1"]) / len(X))
    grads["dT1"][:, 1:] += theta1[:, 1:] * reg / len(X)

    grads["dT2"] += (np.dot(grads["dA3"].T, cache["A2"]) / len(X))
    grads["dT2"][:, 1:] += theta2[:, 1:] * reg / len(X)

    theta_grads = np.concatenate((grads["dT1"].ravel(), grads["dT2"].ravel()))

    return (coste, theta_grads)


def modelo(input_size, num_labels, X, Y, reg, iterations):
    parameters = {}
    inner_layer = 40
    theta = []
    params_rn = np.concatenate((pesosAleatorios(input_size,
inner_layer).ravel(),pesosAleatorios(inner_layer, num_labels).ravel()))

    min = opt.minimize(backprop, params_rn, args=(input_size, inner_layer,
num_labels, X, Y, reg), method='TNC', options={'maxiter': iterations},
jac=True)
    params_rn = min.x
    params_rn = params_rn.reshape(len(params_rn), 1)
    theta1 = np.reshape(params_rn[: inner_layer * (input_size + 1)],
(inner_layer, (input_size + 1)))
    theta2 = np.reshape(params_rn[inner_layer * (input_size + 1) :],
(num_labels, (inner_layer + 1)))
    parameters['theta1'] = theta1
    parameters['theta2'] = theta2
```

```
    return parameters


def choose_lambda_acc(X, Y, X_val, Y_val, C=3):
    lambd = np.array([0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30,
300])
    accuracy_dots = np.zeros(len(lambd))
    accuracy_val = np.zeros(len(lambd))

    for i in range(len(lambd)):
        theta = modelo(X.shape[1], 3, X, Y, lambd[i], 100) #300

        prediction, _ = L_model_forward(X, theta)
        index_max = np.argmax(prediction, axis = 1)
        index_max = index_max.reshape((len(index_max), 1))
        accuracy_dots[i] = np.sum(np.argmax(Y, axis = 1).reshape((len(Y), 1))
== index_max) / Y.shape[0]

        predict_val, _ = L_model_forward(X_val, theta)
        index_max = np.argmax(predict_val, axis = 1)
        index_max = index_max.reshape((len(index_max), 1))
        accuracy_val[i] = np.sum(np.argmax(Y_val, axis =
1).reshape((len(Y_val), 1)) == index_max) / Y_val.shape[0]
    plt.plot(lambd, accuracy_dots)
    plt.plot(lambd, accuracy_val)
    plt.xlabel("lambda")
    plt.ylabel("accuracy")
    plt.title("Searching optimal lambda")
    print(accuracy_dots)
    print(accuracy_val)
    plt.show()


def calculate_probability(X, Y, theta, C=3):
    AL, _ = L_model_forward(X, theta)
    indexes = np.argmax(AL, axis=1)
    index_max = indexes.reshape((len(indexes), 1))
    accuracy = np.sum(np.argmax(Y, axis = 1).reshape((len(Y), 1)) ==
index_max) / Y.shape[0]
    print("Accuracy " + str(accuracy * 100) + '%')
```

```
    precision = np.zeros(C)
    recall = np.zeros(C)
    for c in range(C):
        precision[c] = np.sum((Y[:, c] == 1) * (index_max == c).ravel()) /
np.sum(index_max == c)
        recall[c] = np.sum((Y[:, c] == 1) * (index_max == c).ravel()) /
np.sum(Y[:, c] == 1)
    print("precision dogs: " + str(precision[0]))
    print("precision cats: " + str(precision[1]))
    print("precision elephants: " + str(precision[2]))

    print("recall dogs: " + str(recall[0]))
    print("recall cats: " + str(recall[1]))
    print("recall elephants: " + str(recall[2]))

    print("f1 score dogs: " + str(2 * precision[0] * recall[0] / (precision[0]
+ recall[0])))
    print("f1 score cats: " + str(2 * precision[1] * recall[1] / (precision[1]
+ recall[1])))
    print("f1 score elephants: " + str(2 * precision[2] * recall[2] /
(precision[2] + recall[2])))

    print("average dogs: " + str((precision[0] + recall[0]) / 2))
    print("average cats: " + str((precision[1] + recall[1]) / 2))
    print("average elephants: " + str((precision[1] + recall[1]) / 2))

#choose_lambda_acc(X_train, Y_train, X_dev, Y_dev, C=3)

params = modelo(X_train.shape[1], 3, X_train, Y_train, 30, 300)
print("Accuracy training set:")
calculate_probability(X_train, Y_train, params, C=3)
print("Accuracy validation set:")
calculate_probability(X_dev, Y_dev, params, C=3)
print("Accuracy test set:")
calculate_probability(X_test, Y_test, params, C=3)
```

Run the model executing:

```
python neural_network_sigmoid.py
```

This model uses 40 neurons in the hidden layer.