Cat recognition with a Neural Network

Delgado De la Paz, Alethia Jocelyn
Tecnológico de Monterrey Campus Querétaro
Intelligent Systems Technologies
Querétaro, México
A01273709@tec.mx

***Abstract -** A neural network is a system of multiple layers with interconnected neurons (nodes) that exchange messages between each other. Each connection has a numeric weight that is updated during the training process. Neural networks are used in different areas such as pattern recognition, speech recognition, or video analysis [1]. For this project the purpose is to create a neural network for image recognition, in this case, images of cats.*

***Keywords** - Neural network, hidden layers, image recognition, activation functions,*

I. INTRODUCTION

Image recognition is used to identify images and categorize them in one or several classes. It can be achieved with a Neural Network because they are computing systems designed to recognize patterns [2].

In a neural network, the layers are built up so that the first layer detects a set of patterns in the input, then the second layer detects patterns of the patterns, and so on [1]. Neural Networks have an input layer, one or more hidden layers, and an output layer.

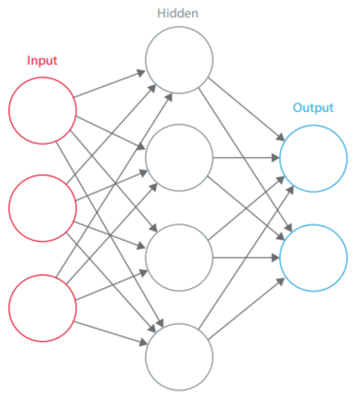

Fig. 1 Example of a neural network

For a Neural Network to recognize different images, first it needs to learn features that identify the object that needs to be identified. Neural Networks learn features from the input data during the training process.

For a binary classification problem, the dataset is labeled with 0 or 1 if it belongs to the desired category. The model then explores the data and learns characteristics for each category. With a Neural Network, the nodes of each successive layer recognize more complex features. The more layers, the better performance the network has. The features learned are chosen by the algorithm.

For this project, a two layer neural network is implemented to identify images of cats.

The methodology followed to implement a Neural Network was:

1. Initialize the parameters (weights and bias) and define the hyperparameters that control the parameters (learning rate, number of iterations)
2. Training loop
   - 2.1. Forward propagation
   - 2.2. Cost function
   - 2.3. Backward propagation
   - 2.4. Update parameters
3. Use the trained parameters to predict labels

II. OBJECTIVE

To create a Convolutional Neural Network model that could distinguish images of cats.

III. DATASET

The dataset used was obtained from Andrew Ng's Neural Networks and Deep Learning course. It comes with two files, a training and a testing dataset. Each data set has the attributes, in this case the images, a column with a 0 or 1 to indicate if it is a cat or not, and a class that also indicates if the picture is a cat picture.

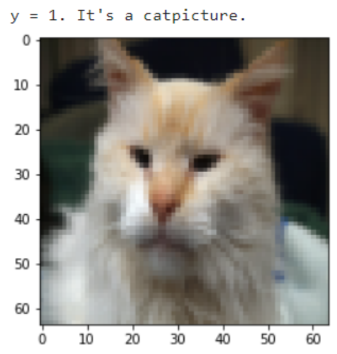

Fig. 2 Example of one sample of the train dataset

The number of training examples is 209 and the number of testing examples is 50. The size of the images is (64, 64, 3).

To use the dataset, each image was reshaped and flattened into a vector size of (12288, 1). First the images are flattened to a vector size (12288, 209) for the train set and (12288, 50) for the test set. The -1 in the .reshape() function flattens the remaining dimensions of the images.

```
train_x_flatten = train_x_orig.reshape
(train_x_orig.shape[0], -1).T
test_x_flatten = test_x_orig.reshape
(test_x_orig.shape[0], -1).T
```

Then the flattened images are divided by 255 to convert the pixel values into a range of 0 to 1.

```
train_x = train_x_flatten / 255.
test_x = test_x_flatten / 255.
```

IV. NEURAL NETWORK

For the Neural Network the size and the input layer, the size of the hidden layer and the size of the output layer were assigned to different values, n_x, n_h and n_y respectively.

According to the methodology to implement a Neural Network, the first step is to initialize the parameters . For this the initialize_parameters() function was defined, and the weights and bias of the two layers were initialized. The weights were defined as an array of zeros with a size of (7, 12288) for the first layer, and (1, 7) for the second. To initialize the weights, the numpy.random.randn() function was used. The bias for the first layer is sized (7, 1) and (1, 1) for the second. The parameters are then stored in a dictionary called parameters.

The numpy.random.seed() function is used to get the same numbers each time the code is runned. And the assert function ensures that the parameters have the correct size.

```
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
```

The second step is the forward propagation, in this step, the values of the previous nodes are taken to calculate the values of the next layer. From now on, a cache variable is used to store different parameters from each step so that they can be used in the next steps.

The forward propagation is done in the linear_activation_forward() function.

```
def linear_activation_forward(A_prev, W, b, activation):
  if activation == "sigmoid":
    Z, linear_cache = linear_forward(A_prev, W, b)
    A, activation_cache = sigmoid(Z)

  elif activation == "relu":
    Z, linear_cache = linear_forward(A_prev, W, b)
    A, activation_cache = relu(Z)

  assert (A.shape == (W.shape[0], A_prev.shape[1]))
  cache = (linear_cache, activation_cache)
  return A, cache
```

For the linear_activation_function, first the linear_forward is defined, where an hypothesis function is calculated. The A is the activation function result from the previous layer, W are the weights, and b the bias. In this step, the A, W, and b are stored in cache.

```
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
```

Then, depending on the activation function of the current layer, the sigmoid or the relu functions are called. These functions can be seen in Annex 1.

The third step is to calculate the cost. The formula to calculate the cost is:

$$J(w,\, b) \;=\; -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} log(\hat{y}^{(i)}) + log(1 - \hat{y}^{(i)})\right]$$

In this function, y_p is the prediction, and Y is the true label (0 or 1)

```
def compute_cost(y_p, Y):
  m = Y.shape[1]

  cost = (-1 / m) * np.sum(np.multiply(Y,
np.log(y_p)) + np.multiply(1 - Y, np.log(1 -
y_p)))

  cost = np.squeeze(cost)
  assert(cost.shape == ())

  return cost
```

The next step is the Backward propagation. Here the Gradient descent is applied to the parameters. First the linear_activation_backward function is called. In this function the activation function corresponding to the layer is called, whether it is sigmoid_backward or relu_backward. These functions are in Annex 2. Then, the result of the activation functions is used as input in the linear_backward function, where the gradient descent formula for each parameter is applied. Here the cache is used to obtain the A, W and b parameters from the previous layer.

```
def linear_backward(dZ, cache):
  A_prev, W, b = cache[0]
  m = A_prev.shape[1]

  dW = np.dot(dZ, A_prev.T) / m
  db = np.squeeze(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)

  return dA_prev, dW, db
```

To update the parameters, the function update_parameters was defined.

```
def update_parameters(parameters, grads,
learning_rate):
 L = len(parameters) // 2

 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
```

All these functions are finally called in the two_layer_model function. This function has the images, the class, learning rate and number of iterations as inputs. The other input is print_cost which prints the cost every 100 iterations.

First, it calls initalize_parameters. Then it enters into the training for loop which runs the amount of iterations indicated. Then, it calls the linear_activation_backward function, for the first layer, it uses the relu activation function, and for the second layer it uses sigmoid. One the Forward propagation is done, it calculates the cost. Then, it does the backward propagation by calling the linear_activation_backward function. Since for the last layer the sigmoid function was used, the back propagation starts with the sigmoid activation function, and then the linear_activation_backward is called again but with the relu function.

After the Backward propagation, the size of the weights and bias need to be the same as they were at the beginning. So the bias was reshaped to make sure the sizes matched.

```
  auxdb1 = db1
  auxdb1 = np.reshape(auxdb1, (7,1))
  auxdb2 = db2
  auxdb2 = np.reshape(auxdb2, (1,1))
  db1 = auxdb1
  db2 = auxdb2
```

V. RESULTS

After 2500 iterations and with a learning rate of 0.007, the cost was 0.05495.

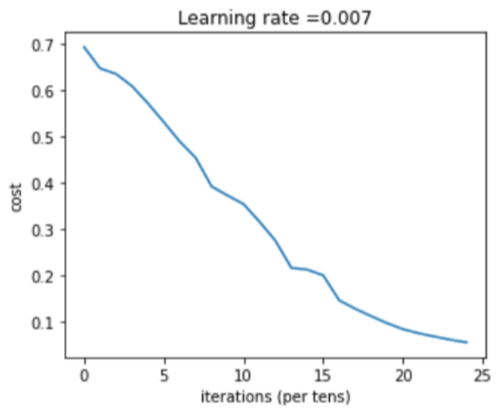

Fig. 3 Plot of the Cost

To calculate the accuracy of the model, two functions were defined. The first one was make_predictions, which is called in the predict function. These functions are in Annex 3.

The accuracy of the model with the training set was: 0.9904

```
predictions_train = predict(train_x, train_y, parameters)
```

```
Accuracy: 0.9904306220095692
```

The accuracy of the model with the testing set was: 0.72

```
predictions_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.72
```

VI. CONCLUSION

Image recognition problems can be easily solved with a neural network. For this project a two layer neural network was applied and the accuracy obtained was admissible. For bigger problems with image recognition, a Convolutional Neural Network with more layers can be applied, since CNNs have been proved to be the best solution with a correct detection rate of 99.77 percent.

If this same neural network wants to be used to make better predictions, the dataset can be augmented by using the data augmentation technique.

VII. REFERENCES

[1] Hijazi, S., Kumar, R., & Rowen, C. (2015). Using convolutional neural networks for image recognition. *Cadence Design Systems Inc.: San Jose, CA, USA*, *9*.

[2] Editor. (2020, January 13). *Image Recognition with Deep Neural Networks and its Use Cases*. AltexSoft. https://www.altexsoft.com/blog/image-recognition-neural-networks-use-cases/

[3] M. (2019, April 14). *Neural Network*. Kaggle. https://www.kaggle.com/code/mriganksingh/neural-network/data

VIII. ANNEXES

Annex 1: Activation functions

```
def sigmoid(Z):
  s = 1 / (1 + np.exp(-Z))
  cache = Z
  return s, cache

def relu(Z):
  r = np.maximum(0, Z)
  assert(r.shape == Z.shape)
  cache = Z
  return r, cache
```

Annex 2: Backward activation functions

```
def sigmoid_backward(dA, activation_cache):
  Z = activation_cache[1]
  s = 1 / ( 1 + np.exp(-Z))
  dZ = dA * s * (1 - s)
  assert(dZ.shape == Z.shape)
  return dZ

def relu_backward(dA, activation_cache):
  Z = activation_cache[1]
  dZ = np.array(dA, copy = True)
  dZ[Z <= 0] = 0
  assert(dZ.shape == Z.shape)
  return dZ
```

Annex 3: Predictions

```
def make_predictions(X, parameters):
  caches = []
  A = X
  L = len(parameters) // 2
  for l in range(1, L):
    A_prev = A

    #Linear RELU
    A, cache = linear_activation_forward(A_prev,
                                         parameters['W' + str(l)],
                                         parameters['b' + str(l)],
                                         activation = 'relu')
    caches.append(cache)

    #Linear SIGMOID
    predictions, cache = linear_activation_forward(A,
                                          parameters['W' + str(L)],
                                          parameters['b' + str(L)],
                                          activation = 'sigmoid')

    caches.append(cache)

    assert(predictions.shape == (1, X.shape[1]))
  return predictions, caches

def predict(X, y, parameters):
  m = X.shape[1]
  n = len(parameters) // 2 # number of layers in the neural network
  p = np.zeros((1,m))

  # Forward propagation
```

```
probability, caches = make_predictions(X, parameters)

# convert probability to 0/1 predictions
for i in range(0, probability.shape[1]):
    if probability[0,i] > 0.5:
        p[0,i] = 1
    else:
        p[0,i] = 0

print("Accuracy: "  + str(np.sum((p == y)/m)))

return p
```