Build a Simple Neural Network using Numpy

Build a single neuron using NumPy for image classification.

Ramesh Paudel May 15, 2020 · 6 min read★

In this article, we will discuss how to make a simple neural network using NumPy.

1. **Import Libraries**

First, we will import all the packages that we will need. We will need ***numpy***, ***h5py*** (for loading dataset stored in H5 file), and ***matplotlib*** (for plotting).

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
```

2. Data Preparation

The data is available in (".h5") format and contain training and test set of images labeled as cat or non-cat. The dataset is available in github repo for download. Load the dataset using the following function:

```
def load_dataset():
    train_dataset = h5py.File('datasets/train_catvnoncat.h5', "r")
    train_x = np.array(train_dataset["train_set_x"][:])
    train_y = np.array(train_dataset["train_set_y"][:])

test_dataset = h5py.File('datasets/test_catvnoncat.h5', "r")
    test_x = np.array(test_dataset["test_set_x"][:])
    test_y = np.array(test_dataset["test_set_y"][:])

classes = np.array(test_dataset["list_classes"][:])

    train_y = train_y.reshape((1, train_y.shape[0]))
    test_y = test_y.reshape((1, test_y.shape[0]))

    return train_x, train_y, test_x, test_y, classes
```

We can analyze the data by looking at their shape.

```
train_x, train_y, test_x, test_y, classes = load_dataset()

print ("Train X shape: " + str(train_x.shape))
print ("Train Y shape: " + str(train_y.shape))

print ("Test X shape: " + str(test_x.shape))
print ("Test Y shape: " + str(test_y.shape))
```

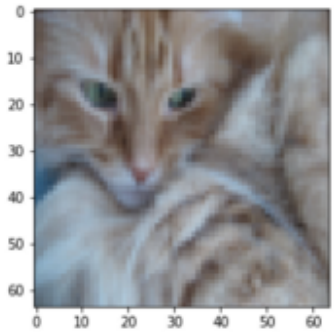
```
Train X shape: (12288, 209)
Train Y shape: (1, 209)
Test X shape: (12288, 50)
Test Y shape: (1, 50)
```

We have 209 train image where each image is square (height = 64px) and (width = 64px) and have 3 channels (RGB). Similarly, we have 50 test images of the same dimension.

Let us visualize the image. You can change the index to see different images.

```
# change index for another image
index = 2
plt.imshow(train_x[index])
```

```
<matplotlib.image.AxesImage at 0x119060cc0>
```



Data Preprocessing: The common data preprocessing for image data involves:

1. Figure out the dimensions and shapes of the data (m_train, m_test, num_px, ...)

2. Reshape the datasets such that each example is now a vector of size (height * width * channel, 1)

3. “Standardize” the data

First, we need to flatten the image. This can be done by reshaping the images of shape (height, width, channel) in a numpy-array of shape (height ∗ width ∗channel, 1).

```
train_x = train_x.reshape(train_x.shape[0], -1).T
test_x = test_x.reshape(test_x.shape[0], -1).T

print ("Train X shape: " + str(train_x.shape))
print ("Train Y shape: " + str(train_y.shape))

print ("Test X shape: " + str(test_x.shape))
print ("Test Y shape: " + str(test_y.shape))
```

```
Train X Flatten shape: (12288, 209)
Train Y shape: (1, 209)
Test X Flatten shape: (12288, 50)
Test Y shape: (1, 50)
```

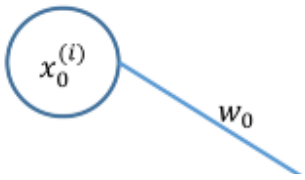
Standardize the data: The common preprocessing step in machine learning is to center and standardize the dataset. For the given picture datasets, it can be done by dividing every row of the dataset by 255 (the maximum value of a pixel channel).

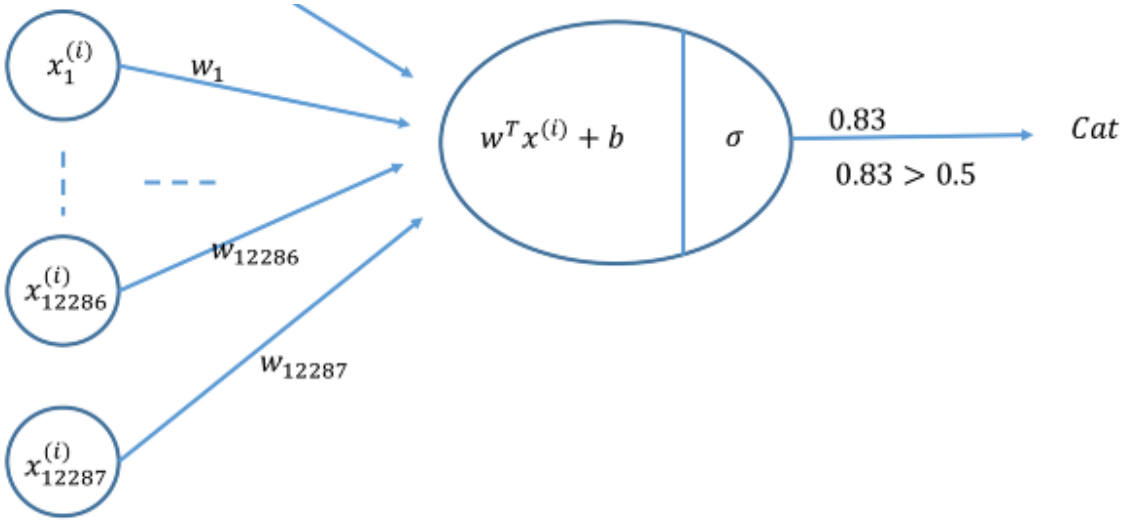
```
train_x = train_x/255.
test_x = test_x/255.
```

Now we will build a simple neural network model that can correctly classify pictures as cat or non-cat.

3. Neural Network Model

We will build a Neural Network as shown in the following figure.





Key steps: The main steps for building a Neural Network are:

1. Define the model structure (like number of input features, number of ouput, etc.)
2. Initialize the model's parameters (weight and bias)
3. Loop:

- Calculate current loss (forward propagation)
- Calculate current gradient (backward propagation)
- Update parameters (gradient descent)

3.1 Activation Function

The sigmoid activation function is given by

$$sigmoid(w^T x + b) = \frac{1}{1+e^{-(w^T x+b)}}$$

The sigmoid activation function can be calculated using *np.exp().*

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

3.2 Initializing Parameters

We need to initialize the parameter w (weight) and b (bias). In the following example, w is initialized as a vector of random numbers using *np.random.randn()* while b is initialize zero.

```
def initialize_parameters(dim):
    w = np.random.randn(dim, 1)*0.01
    b = 0
    return w, b
```

3.3 Forward and Back Propagation

Once the parameters are initialized, we can do the "forward" and "backward" propagation steps for learning the parameters.

- Set of input features (X) are given.
- We will calculate the activation function as given below.

$$A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, \ldots, a^{(m-1)}, a^{(m)})$$

- We will compute the cost as given below.

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

- Finally, we will calculate the gradients as follows (back propagation).

$$\frac{\partial J}{\partial w} = \frac{1}{m} X(A - Y)^T$$

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

$i=1$

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

    #calculate activation function
    A = sigmoid(np.dot(w.T, X)+b)

    #find the cost
    cost = (-1/m) * np.sum(Y * np.log(A) + (1 - Y) * (np.log(1 - A)))
    #find gradient (back propagation)
    dw = (1/m) * np.dot(X, (A-Y).T)
    db = (1/m) * np.sum(A-Y)

    cost = np.squeeze(cost)
    grads = {"dw": dw,
             "db": db}
    return grads, cost
```

3.4 Optimization

After initializing the parameters, computing the cost function, and calculating gradients, we can now update the parameters using gradient descent.

```
def gradient_descent(w, b, X, Y, iterations, learning_rate):
    costs = []
    for i in range(iterations):
        grads, cost = propagate(w, b, X, Y)

        #update parameters
        w = w - learning_rate * grads["dw"]
        b = b - learning_rate * grads["db"]
        costs.append(cost)
        if i % 500 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    params = {"w": w,
              "b": b}
    return params, costs
```

3.5 Prediction

Using the learned parameter *w* and *b*, we can predict the labels for a train or test examples. For prediction we first need to calculate the activation function given as follows.

$$A = \sigma(w^T X + b)$$

Then convert the output (prediction) into 0 (if $\boldsymbol{A}$ <= 0.5) or 1 (if $\boldsymbol{A}$ > 0.5) and store in *y_pred*.

```
def predict(w, b, X):
    # number of example
    m = X.shape[1]
    y_pred = np.zeros((1,m))
    w = w.reshape(X.shape[0], 1)

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

    for i in range(A.shape[1]):
        y_pred[0,i] = 1 if A[0,i] >0.5 else 0
        pass
    return y_pred
```

3.6 Final Model

We can put together all the building block in the right order to make a neural network model.

```
def model(train_x, train_y, test_x, test_y, iterations, learning_rate):
    w, b = initialize_parameters(train_x.shape[0])
    parameters, costs = gradient_descent(w, b, train_x, train_y,
iterations, learning_rate)

    w = parameters["w"]
    b = parameters["b"]

    # predict
    train_pred_y = predict(w, b, train_x)
    test_pred_y = predict(w, b, test_x)

    print("Train Acc: {} %".format(100 - np.mean(np.abs(train_pred_y -
train_y)) * 100))
    print("Test Acc: {} %".format(100 - np.mean(np.abs(test_pred_y -
test_y)) * 100))

    return costs
```

We can use the following code to train and predict on the image dataset using the model built above. We will use the *learning_rate* of 0.005 and train the model for 2000 *iterations*.

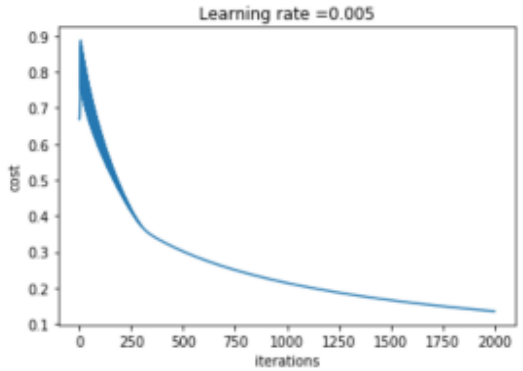
```
costs = model(train_x, train_y, test_x, test_y, iterations = 2000,
learning_rate = 0.005)
```

```
Cost after iteration 0: 0.668697
Cost after iteration 200: 0.461757
Cost after iteration 400: 0.329041
Cost after iteration 600: 0.278076
Cost after iteration 800: 0.241537
Cost after iteration 1000: 0.213692
Cost after iteration 1200: 0.191619
Cost after iteration 1400: 0.173629
Cost after iteration 1600: 0.158657
Cost after iteration 1800: 0.145993
Train Acc: 99.04306220095694 %
Test Acc: 70.0 %
```

Training accuracy is around 99% which means that our model is working and fit the training data with high probability. Test accuracy is around 70%. Given the simple model and the small dataset, we can consider it as a good model.

Finally, we can plot the cost and see how the model was learning parameters.

```
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations')
plt.title("Learning rate =" + str(d["learning_rate"]))
plt.show()
```



We can see the cost decreasing in each iteration which shows that the parameters are being learned.

In the next article, we will discuss how to make a neural with a hidden layer.