Implementing Neuron Network from scratch in Python

How to be useful

May 2023

1 Introduction

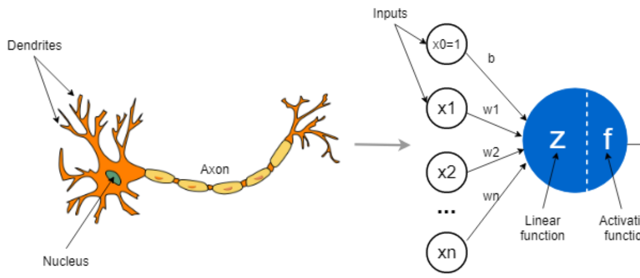
Hello, welcome to my blog. In this article, we will cover how neuron network runs under the hood in theory and apply these to code. After this blog, I hope you will have some intuitions about how neuron networks work.

2 Why neuron networks

As you can see, many applications nowadays like face recognition, vehicle detection, and famous chat-gpt are built with deep learning. Although it uses different architectures but neuron networks are the backbones behind all of these models. Deep learning is a subset of machine learning. Before this sophisticated subset, many data scientists or data researchers used machine learning models, which are also called statistical models, such as Support Vector Machines, Linear Regression, and Decision Trees to find the pattern or the trend and predict the future outcomes of the dataset. These ml models act as a function that maps x to y. (f(x) = y). Where x is data in y is output. But in the era of Big-data. Given a massive dataset, which can be hundreds of gigabytes. These statistical models are not sufficient to capture the pattern of data due to their variety. Data is not only numerical but it can be represented as image, sound, etc. This is where deep learning comes in.

3 Perceptron

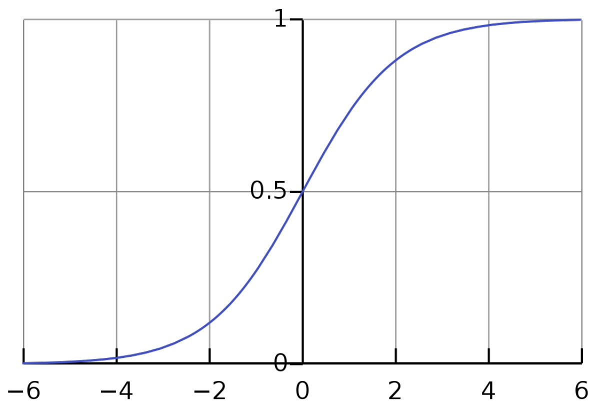
The Perceptron algorithm is a supervised machine-learning model for binary



classification that mimics the neuron unit in our brain.
The function of perceptron describes as a linear function f(x)=wx+b, where w(weight), b(bias) are parameters, x is data input, and g(f(x)), where g(x) is activation function(sigmoid in this case).

$$g(x) = \frac{1}{1 + e^{-x}}$$

With sigmoid activation function. The algorithm will tell us what is the probability that a given data input belongs to this label. The range will be squeezed



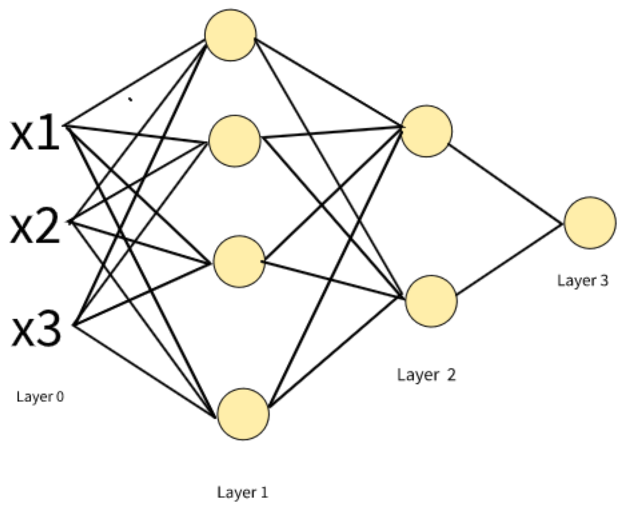
into [0,1] A deep neuron network is a combination of a set of perceptrons at each layer. Each unit represents a perceptron.

3.1 Note

I recommend you search for why perceptron but not other algorithms and why they have to use a non-linearity activation function such as sigmoid after the linear function.

4 Neuron networks

4.1 Notation



In the above picture. L = 3(number of layers). number of examples: m The number of units in layer l

$$n^{[}l]$$

Activation value

$$a^{[}l]$$

Type of activation function

$$g[l]$$

Linear value

$$z^{[}l]$$

Weight and bias

$$W^{[}l], b^{[}l]$$

4.2 Foward propagation

In this section, we will cover forward propagation and backpropagation in neuron networks. From the input to the prediction(output). At layer 1:

$$z^{[}1] = w^{[}1] * a^{[}0] + b^{[}1]$$

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

At layer 2:

$$z^{[}2] = w^{[}2] * a^{[}1] + b^{[}2]$$

$$a^{[}2] = g^{[}2](z^{[}2])$$

At layer 3:

$$z^[3] = w^[3] * a^[2] + b^[3]$$

$$a^[3] = g^[3](z^[3])$$

To summarize, at layer l:

$$z^[l] = w^[l] * a^[l-1] + b^[l]$$

$$a^[l] = g^[l](z^[l])$$

Shape:

$$z^[l], a^[l], b^[l] = (n^[l], m)$$

$$w^[l] = (n^[l], n^[l-1])$$

$$b^[l] = (n^[l], 1)$$

For example: If we have 250 examples in an iteration, then the shapes are: m=250,

$$w^[1] = (4,3), w^[2] = (2,3), w^[3] = (1,2)$$

$$a^[0] = (250,3), a^[1] = (4,250), a^[2] = (2,250), a^[3] = (1,250)$$

Each layer l in g[l] will have an activation function type. In this case, layer 1,2 will be ReLU activation function, and the last layer will depend on your output. If binary classification, then use the sigmoid activation function, if multi-label classification, then use the softmax activation function.

a[L] in the last layer L is the prediction(that).

4.3 Cost function

A cost function(loss function, objective function) is a function that measures how good the predicted is with the true labels. During training, we want this cost as small as possible. In this case, I will use binary cross-entropy as a loss function.

$$J = L(y^{hat}, y) = L(a^[L], y) = \frac{-1}{m} * \sum_{i=1}^{m}(y_i * log(a_i^[L]) + (1-y_i) * log(1-a_i^[L]))$$

We use this equation to tune the parameters(w,b)

4.4 Backpropagation

In this section, we will use backpropagation to minimize the loss function by tuning the parameters. This section will be more understandable if you know the chain rule, the derivative of a function. From the forward pass, we got

$$a[L]$$

, we would like to find the derivative w.r.t(with respect to)

$$w^{[}l], b^{[}l]$$

from l=1...3. From the chain rule, we got:

At layer 3:

$$\frac{\partial J}{\partial w^{[}3]} = \frac{\partial J}{\partial a^{[}3]} * \frac{\partial a^{[}3]}{\partial z^{[}3]} * \frac{\partial z^{[}3]}{\partial w^{[}3]}$$

$$\frac{\partial J}{\partial b^{[}3]} = \frac{\partial J}{\partial a^{[}3]} * \frac{\partial a^{[}3]}{\partial z^{[}3]} * \frac{\partial z^{[}3]}{\partial b^{[}3]}$$

At layer 2:

$$\frac{\partial J}{\partial w^{[}2]} = \frac{\partial J}{\partial z^{[}3]} * \frac{\partial z^{[}3]}{\partial a^{[}2]} * \frac{\partial a^{[}2]}{\partial z^{[}2]} * \frac{\partial z^{[}2]}{\partial w^{[}2]}$$

$$\frac{\partial J}{\partial b^{[}2]} = \frac{\partial J}{\partial z^{[}3]} * \frac{\partial z^{[}3]}{\partial a^{[}2]} * \frac{\partial a^{[}2]}{\partial z^{[}2]} * \frac{\partial z^{[}2]}{\partial b^{[}2]}$$

At layer 1:

$$\frac{\partial J}{\partial w^{[}1]} = \frac{\partial J}{\partial z^{[}2]} * \frac{\partial z^{[}2]}{\partial a^{[}1]} * \frac{\partial a^{[}1]}{\partial z^{[}1]} * \frac{\partial z^{[}1]}{\partial w^{[}1]}$$

$$\frac{\partial J}{\partial b^{[}1]} = \frac{\partial J}{\partial z^{[}2]} * \frac{\partial z^{[}2]}{\partial a^{[}1]} * \frac{\partial a^{[}1]}{\partial z^{[}1]} * \frac{\partial z^{[}1]}{\partial b^{[}1]}$$

Do not panic, we will analyze it. We will start from layer 3(the last layer)

$$z^{[}3] = w^{[}3] * a^{[}2] + b^{[}3]$$

and then:

$$a^{[}3] = g^{[}3](z^{[}3])$$

$$(a^{[}3] = a^{[}L])$$

) and then:

$$J = L(a^{[}L], y) = \frac{-1}{m} * \sum_{i=1}^{m} (y_i * log(a_i^{[}L]) + (1 - y_i) * log(1 - a_i^{[}L]))$$

Moving backward, we got: 1/

$$\frac{\partial J}{\partial a^{[}3]} = \frac{1}{m} * (\frac{1-y}{1-a^{[}3]} - \frac{y}{a^{[}3]})$$

2/

$$\frac{\partial a^{[}3]}{\partial z^{[}3]} = g^{'}(z^{[}3]) * z^{[}3]$$

3/

$$\frac{\partial z^{[}3]}{\partial w^{[}3]} = (a^{[}2])$$

Combine 1,2,3, and we got:

$$\frac{\partial J}{\partial w^[3]} = \frac{\partial J}{\partial a^[3]} * \frac{\partial a^[3]}{\partial z^[3]} * \frac{\partial z^[3]}{\partial w^[3]}$$

You can see it if you know the derivative. ;) I will write under this:

$$\frac{\partial J}{\partial w^[l]} = dw^[l]$$

$$\frac{\partial J}{\partial b^[l]} = db^[l]$$

$$\frac{\partial J}{\partial z^[l]} = dz^[l]$$

$$\frac{\partial J}{\partial a^[l]} = da^[l]$$

To summarize, here is what happens when backpropagation at layer l:

$$dz^[l] = da^[l] * g^{'[l]}(z^[l])$$

$$dw^[l] = dz^[l] * a^[l-1].T$$

$$db^[l] = dz^[l]$$

$$da^[l-1] = w^[l].T * dz^[l]$$

This is the same for all hidden layers except for the final layer. if we use binary-cross entropy then:

$$\frac{\partial J}{\partial a^[L]} = \frac{1}{m} * (\frac{1-y}{1-a^[L]} - \frac{y}{a^[L]})$$

if we use cross-entropy(multiclass classification) then:

$$\frac{\partial J}{\partial a^[L]} = \frac{1}{m} * (\frac{1}{1-a^[L]} * Y)$$

4.5 Update rule

Recall from backpropagation, for each layer, we got $dw^[l], db^[l]$, we will use gradient descent to update parameter $w^[l], b^[l]$

$$w^[l] = w^[l] - \eta * dw^[l]$$

$$b^[l] = b^[l] - \eta * db^[l]$$

Where

$$\eta$$

means learning rate.

In this step, we are done with one iteration of our training process. An iteration is also called an epoch. It means you go through all of the training examples in your dataset.

5 Implement neuron networks in Python

The code that I have written is very similar to the code from the programming exercise in week 4 course 1 in the Deep learning specialization taught by Andrew Ng on Coursera.

5.1 Initialize parameters

In the first step, we will initialize parameters for weight, and bias. This step is important since it affects the convergence speed. One of the most common initialization is 'He initialization'. But I will use random initialization in this article.

```
import numpy as np
import matplotlib.pyplot as plt
def initialize_parameters(layer_dims):
    parameters = {}
    L = len(layer_dims) # number of layers in the network

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

layerdims will be a list consist the number of units in each layer. For the above example, we got layersdims=[3,4,3,2,1] The function returns initialized parameters.

5.2 Feed forward

```
def linear_activation_forward(A_prev, W, b, activation):
    Z, linear_cache = forward_pass(A_prev, W, b)
    if activation == 'sigmoid':
        A, activation_cache = sigmoid(Z)
    if activation == 'relu':
        A, activation_cache = relu(Z)
    cache = (linear_cache, activation_cache)
    return A, cache
```

The first line of this function perform

$$z^[l] = f(a^[l-1]) = w^[l] * a^[l-1] + b^[l]$$

```
def forward_pass(A, W, b):
    Z = np.dot(W, A) + b
    cache = (A, W, b)
    return Z, cache
```

The reason why we need the variable 'cache' is to save for backpropagation(since we need $a^[l-1], z^[l]$ Sigmoid and relu are two activation functions $g(f(a^[l-1]))$

```
def sigmoid(Z):
    return 1/(1+np.exp(-Z)), Z
def relu(Z):
    return np.maximum(Z, 0), Z
```

activation cache is $z^[l]$
To sum up, the variable cache for each layer comprises of $1/a^[l-1], w^[l], b^[l]$: 1st variable $2/z^[l]$ -¿ 2nd variable Combine all of these, we got:

```
def L_model_forward(X, parameters):
    caches = []
    A = X
    L = len(parameters) // 2
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev, parameters['W'+str(l)], parameters['b'-
        caches.append(cache)
    AL, cache = linear_activation_forward(A, parameters['W'+str(L)], parameters['b'+str(L)]
    caches.append(cache)
    return AL, caches
```

AL is y^{hat}, the prediction, caches are list consisting $a^[l-1], z^[l]$ for each layers

5.3 Cost function

We will follow this equation $J = L(y^{hat}, y) = L(a^[L], y) = \frac{-1}{m} * \sum_{i=1}^{m}(y_i * log(a_i^[L]) + (1-y_i) * log(1-a_i^[L]))$ to implement.

```
def compute_cost(AL, Y):
    m = Y.shape[1]
    cost = (-1/m) * np.sum(np.multiply(Y, np.log(AL)) + np.multiply(1-Y, np.log(1-AL)))
    cost = np.squeeze(cost)
    return cost
```

5.4 Back-propagation

```
def linear_activation_backward(dA, cache, activation):
    linear_cache, activation_cache = cache
    if activation == 'relu':
        dZ = relu_backward(dA, activation_cache)
    if activation == 'sigmoid':
        dZ = sigmoid_backward(dA, activation_cache)
    dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

For each layer in backpropagation, this will return $dA_prev, dW^[l], db^[l]$

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

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

These two function above will compute this $dz^[l] = da^[l] * g'^{[l]}(z^[l])$. $g'^{[l]}(z^[l])$ is the derivative of activation function(sigmoid, relu) Having dZ, we will compute dw, db, $da^[l-1]$

```
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)
```

$$dw^[l] = dz^[l] * a^[l-1].T$$

$$db^[l] = dz^[l]$$

$$da^[l-1] = w^[l].T * dz^[l]$$

Since we compute it with m example, dz will have shape (n[l], m), so we have to sum all dZ in order to db will be (n[l], 1), which is the same size of b. Combine all of this, we got:

```
def L_model_backward(AL, Y, caches):
    grads = {}
    L = len(caches)
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)

    dAL = - (np.divide(Y,AL) - np.divide(1-Y, 1-AL))
    current_cache = caches[-1]

    dA_prev_temp, dW_temp, db_temp = linear_activation_backward(dAL, current_cache, 'sigmoi

    grads['dA'+str(L-1)] = dA_prev_temp
    grads['dW'+str(L)] = dW_temp
    grads['db'+str(L)] = db_temp

    for l in reversed(range(L-1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads['dA'+str(l+1)], cu

        grads['dA'+str(l)] = dA_prev_temp
        grads['dW'+str(l+1)] = dW_temp
        grads['db'+str(l+1)] = db_temp
    return grads
```

grads is a dictionary that contains dW, db for each layer, except the first layer.

5.5 Update parameters

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

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

We code follow these equations:

$$w^[l] = w^[l] - \eta * dw^[l]$$

$$b^[l] = b^[l] - \eta * db^[l]$$

5.6 Combine all of this

```
def L_layer_model(X, Y, layer_dims, learning_rate = 0.001, num_iterations = 3000, print_cost
    costs = []
    parameters = initialize_parameters(layer_dims)

    for i in range(0, num_iterations):
        AL,  caches = L_model_forward(X, parameters)
        cost = compute_cost(AL, Y)
        grads = L_model_backward(AL, Y, caches)

        parameters = update_parameters(parameters, grads, learning_rate)

        if print_cost and i % 1000 == 0 :
            print(cost, f'at iter {i}')
            costs.append(cost)
    return parameters, cost
```

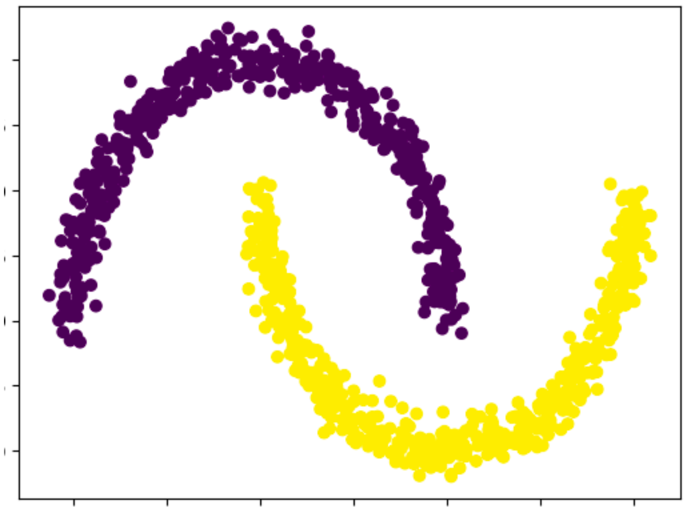
I also add prediction function which return 1 when AL ¿ 0.9 and acc function to return percentage accuracy

```
def predict(X, params):
    AL,  caches = L_model_forward(X, params)
    l = np.zeros_like(AL)
    l[AL>0.9]=1
    return l
def acc(predict, label):
    y_hat = np.squeeze(predict)
    label = np.squeeze(label)
    c = 0
    for x in range(len(y_hat)):
        if int(y_hat[x]) == label[x]:
            c+=1
    return c/y_hat.shape[0]
```

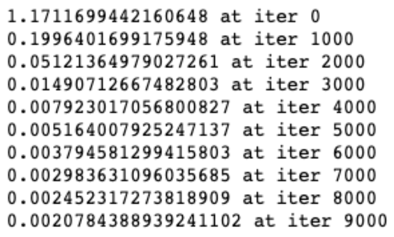
6 Toy data

We will apply above model to make moon dataset from sklearn

```
from sklearn.datasets import make_moons
features, true_labels = make_moons(n_samples = 1000, noise = 0.05)
plt.scatter(features[:,0], features[:,1], c=true_labels)
```



```
X = features
Y = np.expand_dims(true_labels, axis=0)
layer_dims = [2,4,4,1]
params, cost = L_layer_model(X.T, Y, layer_dims, learning_rate = 0.1, num_iterations = 1000
```

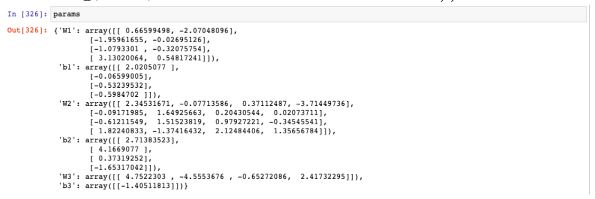


```
1.1711699442160648 at iter 0
0.1996401699175948 at iter 1000
0.05121364979027261 at iter 2000
0.01490712667482803 at iter 3000
0.007923017056800827 at iter 4000
0.005164007925247137 at iter 5000
0.003794581299415803 at iter 6000
0.002983631096035685 at iter 7000
0.002452317273818909 at iter 8000
0.0020784388939241102 at iter 9000
```

Finally, we make prediction:

```
y_hat = predict(X.T, params)
acc(y_hat, Y)
```

The result is amazing, 1.0, this must be overfited :)) You can see parameters after trained:



```
In [326]: params

Out[326]: {'W1': array([[ 0.66599498, -2.07048096],
                 [-1.95961655, -0.02695126],
                 [-1.0793301 , -0.32075754],
                 [ 3.13020064,  0.54817241]]),
           'b1': array([[ 2.0205077 ],
                 [-0.06599005],
                 [-0.53239532],
                 [-0.5984702 ]]),
           'W2': array([[ 2.34531671, -0.07713586,  0.37112487, -3.71449736],
                 [-0.09171985,  1.64925663,  0.20430544,  0.02073711],
                 [-0.61211549,  1.51523819,  0.97927221, -0.34545541],
                 [ 1.82240833, -1.37416432,  2.12484406,  1.35656784]]),
           'b2': array([[ 2.71383523],
                 [ 4.1669077 ],
                 [ 0.37319252],
                 [-1.65317042]]),
           'W3': array([[ 4.7522303 , -4.5553676 , -0.65272086,  2.41732295]]),
           'b3': array([[-1.40511813]])}
```

7 End

For the above example, I use layerdims[2,4,4,1] for 2 input feature, 2 hidden layers, each hidden layers have 4 units, and 1 output. You can change the number of hidden units and hidden layers if you want. Thank you for reading unitl this line, of course there are a lot of thing to implement with the above model. I hope you can gain some intuitions about deep neuron networks