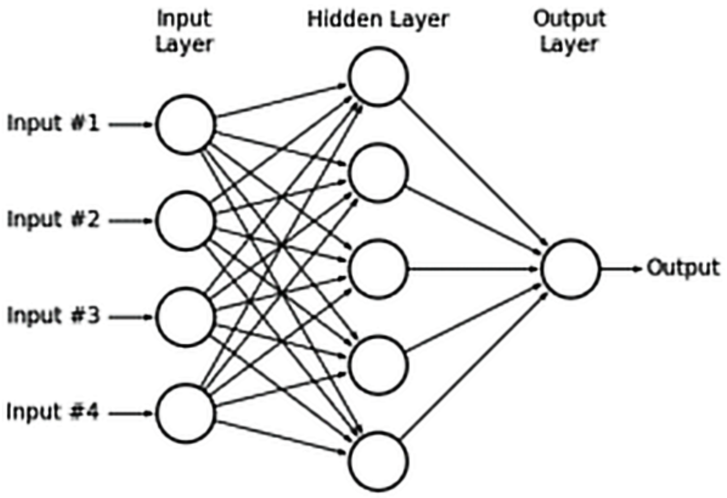
Assignment 2
Neural Networks

1 Introduction

In this assignment you will be building a fully working neural network from scratch. A neural network as you know consists of

- Input layer.
- Multiple hidden layers.
- Output layer.



As shown in the figure above, each layer consists of multiple perceptron. The input dimension of each layer is determined by the number of input nodes constructing that layer.

The weights of the layers are the main learning parameter for any neural networks. Each layer will have weights equal to the input dimension of that layer multiplied by the output dimension of the previous layer. The only exception is the input layer as it has no previous layers so no weights are needed.

To build a neural networks from scratch, multiple helper functions will be implemented, each helper function you are required to implement will have detailed explanation of how it should works, the desired input, and output.

Each part will be explained and implemented step-by-step.Please stick to the function names, input parameters and output parameters

2 Forward Pass

2.1 Parameters initialization

Function Name: *initialize_parameters(linear_dims)*
Input: *linear_dims* , A list containing the dimensions of our neural network.
Return: *parameters* python dictionary containing learnable parameters, for example

- parameters['W' + str(l)] = ...
- parameters['b' + str(l)] = ...

where l is the layer number
This is the first helper function that will be needed for our implementation. This function will take as input a list of integers, those integers will represent the layers dimensions.
Example 1:
If our network consists of 4 layer: 1 input layer of size 3, 1 hidden layer of size 4 , 1 hidden layer of size 2 and 1 output layer of size 3. Then the input of the parameters initialization function will be equal to = [3 , 4, 2 , 3] From the input you can realise two important things

- Depth of the network (number of layers).
- The weights shape of each layer.

For each layer you will have to initialize

- a **random** weight matrix with shape (n_l, n_{l-1})
- a **zero** bias vector with shape $(n_l, 1)$

where n_l represents the dimension of the current layer and n_{l-1} represents the dimension of the previous layer.

You will need to return a python dictionary where each key representing either a weight matrix or a bias vector.
Example 2
We have 4 layers with dimensions = [3, 4 , 2 , 3], The first layer (layer_0) is an input layer so no weights will be calculated for it.

Starting from the second layer_1:
-Weight matrix shape should be (4 , 3)
-bias a vector (4 , 1)

Third layer layer_2:
- Weight matrix shape should be (2 , 4)
- Bias a vector (2 , 1)

Output layer layer_3:
- Weight matrix shape should be (3 , 2)
- Bias a vector (3 , 1)

So dictionary output will be :

$\{"W\_1" : matrix(4, 3),$
$"b\_1" : matrix(4, 1),$
$"W\_2" : matrix(2, 4),$
$"b\_2" : matrix(2, 1),$
$"W\_3" : matrix(3, 2),$
$"b\_3" : matrix(3, 1),$
$\}$

2.2 Linear Forward

Function Name: $linear\_forward(A, W, b)$

Input:

- A, a matrix containing the activation values of the previous layer (l-1)
- W, a matrix containing the weights values of the current layer (l)
- b, a matrix containing the bias values of the current layer (l)

Return:

- Z, A matrix contains the output of forward pass
- *cache*, A tuple containing The activation matrix of previous layer, weight matrix and bias vector

Now that you have initialized your parameters, you will do the forward propagation module.

The calculation should be coded in a vectorized way (as explained in the lecture) **in only one line**

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$

and then return both Z and $cache = (W^{[l]}, A^{[l-1]}, b^{[l]})$

2.3 Activation Forward

Function Name: $activation\_forward(A_{prev}, W, b, activation)$
Input:

- A_{prev} , Activation matrix of the previous layer
- W, Weight matrix of the current layer
- b, Bias matrix of the current layer
- *activation*, String with type of activation "relu" or "sigmoid"

Return:

- A Matrix contains the output Activation of forward pass
- *cache* Tuple contains both caches of $linear\_forward$ and $activation\_forward$

There are two types of activation function that we will use in this function *Sigmoid* and *ReLU*.

$$Sigmoid = \frac{1}{1+e^{WA+B}}$$

$$ReLU = max(0, Z)$$

Use the $linear\_forward$ **function previously implemented** to calculate the Z matrix, the forward pass function will also return *cache*, so you will need to save it in a variable called *linear_cache*
Then based on the type of the activation passed in the activation variable apply one of the following activation functions **by importing it from the utils.py file.**.

Hint : to import functions from another file, all you need to do is to write the following line:

```
from file.py import functionA , functionB , functionC
```

The activation function will return the activation matrix A and cache *activation_cache*. After receiving both the *activation_cache* and the *linear_cache* create a tuple called cache $cache = (linear\_cache, activation\_cache)$, and then return both A and *cache*

2.4 Model Forward

Function Name: $model\_forward(X, parameters)$
Input:

- X, a matrix containing the input (assume one sample).
- *parameters*, a dictionary , **output of the** *initialize_parameter* **function**

Return:

- AL activation matrix of the last layer
- *cache* the cache list of each layer (output of the $activation\_forward$)

By now we have implemented the full forward sequence where we take the weights and multiply them by the output of the previous layer and then apply the activation function on them. This should be expanded for all the layers found in the model.

Your are required to calculate a full forward pass using the **activation_forward** functions you implemented and looping over all the layers. You have the freedom of using the any of the two activation functions (relu and sigmoid) for the hidden layers. But you must use the sigmoid activation function for the output layer, in the end return AL, the last activation matrix you calculated, and *caches* the list of caches of all the layers returned from *activation_linear* function.

2.5 Compute Cost

Function Name: $compute\_cost(AL, Y)$
Input:

- AL , The activation value of the output layer
- Y , The ground truth labels

Return: *cost* cross entropy cost.
Now you will need to implement the last function used in the forward propagation, the *compute_cost* function. This needed to be calculated to check whether the model is actually learning or not.

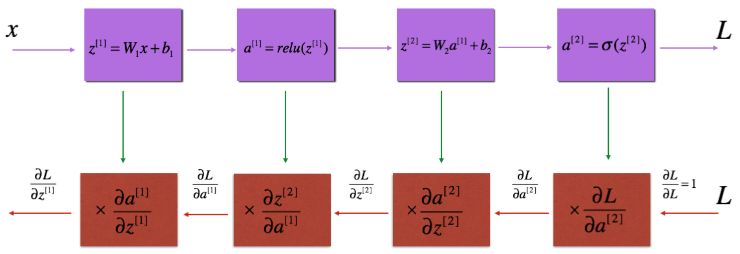
the cost **J** formula that you will be implementing:

$$J = -\frac{1}{m} Y log(AL^T) + (1 - Y) log((1 - AL)^T)$$

The cost computation should be done in the vectorized form, **no loops, just one line of code**, and in the end return **J** value.

3 Backward Pass

Now that we have implemented the forward pass , the backward pass is when we start calculating the gradients for each parameter.


x
$z^{[1]} = W_1 x + b_1$
$a^{[1]} = relu(z^{[1]})$
$z^{[2]} = W_2 a^{[1]} + b_2$
$a^{[2]} = \sigma(z^{[2]})$
L
$\frac{\partial L}{\partial z^{[1]}}$
$\times \frac{\partial a^{[1]}}{\partial z^{[1]}}$
$\frac{\partial L}{\partial a^{[1]}}$
$\times \frac{\partial z^{[2]}}{\partial a^{[1]}}$
$\frac{\partial L}{\partial z^{[2]}}$
$\times \frac{\partial a^{[2]}}{\partial z^{[2]}}$
$\frac{\partial L}{\partial a^{[2]}}$
$\times \frac{\partial L}{\partial a^{[2]}}$
$\frac{\partial L}{\partial L} = 1$
L

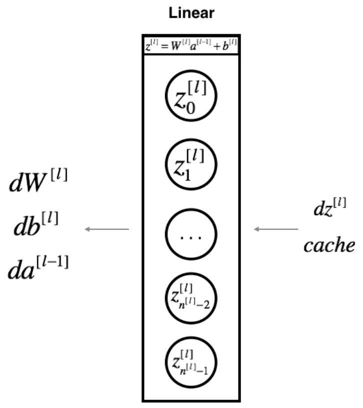
3.1 Linear Backward

Function Name: *linear_backward*$(dZ, cache)$
Input:

- dZ , The derivative of the current layer output
- $cache$, the *linear_cache* variable containing the (A_{prev}, W, b)

Return:

- dA_{prev} the derivative of previous layer Activation matrix.
- dW current layer Weight matrix.
- db derivative of current layer bias.



You should first extract cache tuple to get the A_{prev}, W, b.
$\mathrm{A}_{prev}, W, b = cache$

For layer l, the linear part is: $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$, Suppose you have already calculated the derivative $dZ^{[l]} = \frac{\partial L}{\partial z^l}$. You want to get $(\partial W^{[l]}, \partial b^{[l]}, \partial A^{[l-1]})$.

The three outputs $(\partial W^{[l]}, \partial b^{[l]}, \partial A^{[l-1]})$ are computed using the input $Z^{[l]}$. Here are the formulas you need:

$$dW^{[l]} = \frac{\partial J}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} dA^{[l-1]T}$$

$$db^{[l]} = \frac{\partial J}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$

$$dA^{[l-1]} = \frac{\partial L}{\partial A^{[l-1]}} = W^{[L]T} dZ^{[l]}$$

Each derivative should be calculated in **the vectorized form (no loops) , just one line of code for each derivative**

3.2 Activation Backward Pass

Function Name: $activation\_backward(dA, cache, activation)$
Input:

- dA the derivative of activation output of the current layer
- $cache$ the cache of the current layer containing($linear\_cache$,$activation\_cache$)

Return:

- dA_{prev} the derivative of previous layer Activation matrix.
- dW current layer Weight matrix.
- db derivative of current layer bias.

To help you implement *activation_backward*, we provided two backward functions:

sigmoid_backward: Implements the backward propagation for SIGMOID unit. You can call it as follows:
$dZ = sigmoid\_backward(dA, activation_cache)$

relu_backward: Implements the backward propagation for RELU unit. You can call it as follows:
$dZ = relu\_backward(dA, activation\_cache)$

to access both of those function you have to also import them from the utils.py provided.
After calculating the ∂Z you then shall use the linear_backward previously implemented to calculate the $\partial A_{prev}, \partial W, \partial b$

4 Training

In the last section you will be provided by two helper functions

- $model\_backward()$
- $update\_parameter()$

And a dataset consiting of 97 training samples, the samples represent 2 types of wine each with 13 feature, and a label (0/1) .
You are required to build a training loop with all the functions that you have implemented/provided to train a classifier for this dataset.

You have the freedom to chose the number of layers in you neural network, the dimension of each layer, number of epochs and learning rate to achieve the lowest cost.

Plot the cost of each epoch.

5 Notes

5.1 Deliverables

- Your well commented code in a zip file with the file named ID.zip
- A report showing your work,
  including:
  - Explanation of each function implemented
  - Detailed explanation of both functions "$Backward\_model()$ and $update\_parameters()$"

5.2 Further Notes

- You must use Python
- Copied assignments will be severely penalized.
- You can work in groups of 2

Good Luck