Week-5

Ginjala Lohith Reddy

June 29, 2023

1 What I have learnt

I have learnt many things in Week-5. I have learnt what are sigmoid, relu functions and how they are used to add non linearity to our datasets. I have learnt about the use of new terminology like **weights**, **biases**, **neurons**, **forward and backward models**, learning rate and many more. I have what is Single Layer Neural Network, N-Layered Neural Network and what are uses of this Neural Networks in real world applications. I have also learnt what are Convolution Neural Networks(**CNN**) like how they are better, efficient compared to **ANN**, their application in real world. But this week We have implemented Deep Neural Network from scratch.

2 Implementation

This week as said above we have written code for model of Deep Neural Network from scratch. The code description is as follows, Firstly, We have taken **make_moons** dataset and assigned **1500** elements(X, Y) for training and rest **500** for testing the model. We have found the testing accuracy from Logistic Regression and we have obtained it as **86.6**. Now our aim is to write code for new model with Neural Network that will enhance the testing accuracy compared to Logistic Regression.

2.1 Sigmoid and ReLU

We have written function **sigmoid()** and **relu()** according to their definition(also works for inputs that are matrices) and we can see it in the code. Next we have written function **sigmoid_backward()** and **relu_backward()** for backward model implementation. We have implementated them from derivative of **sigmoid**, **relu** functions and **dJdA**.

2.2 Weights and Biases

Now we have created lists **weights** and **biases** for our model and stored random values from Normal Distribution for weights and zeroes in biases according to shapes of matrices and vectors mentioned in description respectively using **NumPy** library. For **N-Layered** Neural Network we get **N-1** elements in weights and biases.

2.3 Forward Propagation Model

We then have written function **forward_model()** with the help of some small functions **linear_forward()** and **activation_foward()**. Linear function will take input as A_prev, W, b and returns **Z**, **cache** where (Z = W*A_prev + b, * denotes matrix multiplication and cache = (A_prev, W, b)). Similarly activation function introduces non_linearity to our Z which is either **sigmoid** or **relu**. Depending on input parameter actiavtion we proceed with that type of activation and return the value after non-linearity is introduced and total cache(which is combination of linear cache and Z). Now we introduce new variable called cache_backprop which stores cache at every layer and now iterate through N-1 layers with activation as **relu** and N^{th} layer with activation as **sigmoid**. Finally we return the Z obtained after all layers are completed and cache_backprop.

2.4 Cross Entropy Cost Function

We now compute our cost function as given in the description using the final value obtained from forward model and training values of Y. The cost function is similar to cost function of Logistic Regression but we iterate through all examples of (X, Y) and take the mean of all and return 2 decimal rounding of cost.

2.5 Backward Model

Now we have reached the toughest part of whole assignment writing the backward propagation and storing the derivative values with respect to A_prev, W, b respectively. We have 2 helper functions they are **linear_backward()** and **activation_backward()** with which we compute the derivatives. In Linear part we are given the **dJdZ** and **linear_cache**(From which we extract A_prev, W, b) and use the values extracted to compute other paramater derivatives from given description. We now return **dJdA_prev**, **dJdW** and **dJdb**. In similar fashion we introduce sigmoid_backward() and relu_backward() functions to get correct derivative at all layers by removing non-linearity. Now we initialise 3 lists for storing derivatives dJdA, dJdW and dJdb at each layer. We now iterate backwards from N^{th} Layer to 1^{st} Layer and use cache_backprop variable which was computed in forward_model() function to get correct derivatives at all layers. After iterating we return 3 lists.

2.6 Updating Paramters(Weights and Biases)

This function updates weights and biases at every iteration model is learning about dataset and it uses a new variable called learning rate which gives us the rate at which model is learning the data. Depending on Learning rate we can vary the rate of learning of model. We change the weights and biases by their respective derivatives(**dJdW**, **dJdb**) at each layer multipled with learning rate.

2.7 Neural Network Model

This code implements a basic gradient descent algorithm for training a neural network model. It consists of several steps:
Initialization: The code begins by initializing the weights and biases using the initialize_parameters function. The n_neurons parameter specifies the number of neurons in the network.
Gradient Descent Loop: The code then enters a loop that iterates num_iterations(Here in this case it is 3000) times. This loop performs the following steps in each iteration:

1. **Forward Pass:** It computes the output of the model (denoted as A_final) and caches intermediate values required for **Backpropagation** using the **forward_model** function. This step propagates the input data X through the network.

2. **Cost Computation:** It calculates the cost of the model's predictions compared to the actual labels Y using the cross_entropy_cost function. This cost represents how well the model is performing.

3. **Backward Pass:** It computes the gradients of the cost function with respect to the model's outputs (dJdA) and parameters (dJdW and dJdb) using the backward_model function. This step calculates the sensitivity of the cost to changes in the parameters.

4. **Parameter Update:** It updates the weights and biases using the update_parameters function. The gradients (dJdW and dJdb) and a specified learning rate are used to update the parameters. The learning rate controls the step size in the parameter space during optimization.

5. **Cost Printing:** If the print_cost flag is set to True and the current iteration is a multiple of 100, it prints the cost of the model. This allows monitoring the cost value during training.

6. **Return:** Finally, the code returns the updated weights and biases obtained after the gradient descent loop.

2.8 Accuracy of Model

Now we call **model()** function and stores weights and biases, with these parameters and input X_test we compute the Y_pred(which is our prediction) using forward_model() function and compare it with actual Y_test values and find out our model accuracy and prints it.