|  |  |
| --- | --- |
| **Ex No: 3**  **Date:21/08/2024** | **Deep Neural Network Application** |

**Objective:**

To build and train a deep neural network for image classification tasks using Python and the TensorFlow/Keras framework, exploring the implementation of forward and backward propagation, parameter initialization, and model optimization.

To implement a deep neural network with multiple layers from scratch, optimizing it through backpropagation and gradient descent, and applying various activation functions to enhance its classification capabilities.

**Descriptions:**

The model was developed in the previous lab session ([Record](https://github.com/FMS07/Fundamentals_of_DL/blob/main/Lab03-%20Build%20A%20Deep%20Neural%20Network/Lab03-%20Building_Deep_NN-%20Lab_Record.docx)). This section focuses on the functions and detailed implementation of the Deep Neural Network.

**Introduction to Deep Learning and Image Classification:**

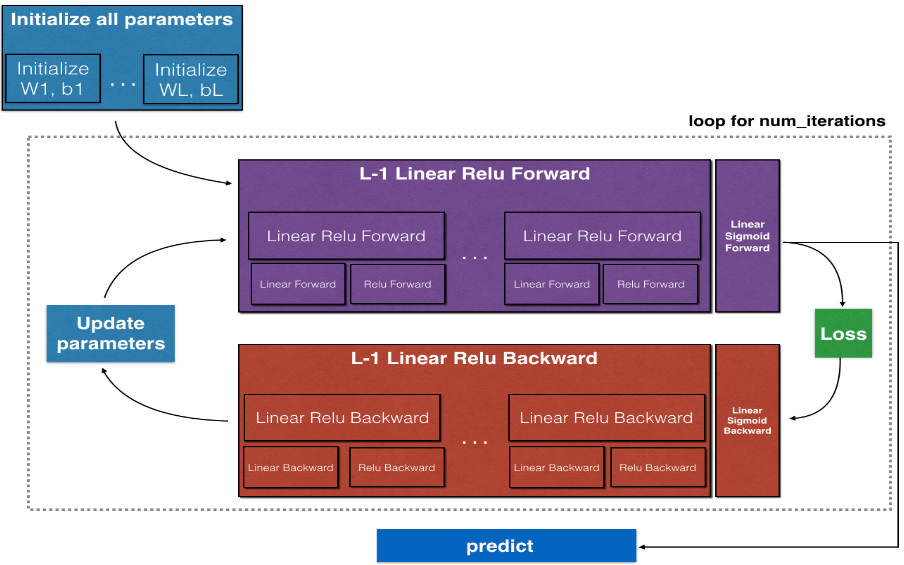
Deep learning involves training neural networks with multiple hidden layers to capture complex patterns in data. Image classification is a common application where a model is trained to recognize objects, animals, or scenes within images. This experiment demonstrates building a deep neural network using multiple layers to classify images into predefined categories.

**Neural Network Architecture:**

This experiment involves constructing a deep neural network with the following components:

* **Input Layer:** Receives the input features (e.g., pixel values of an image).
* **Hidden Layers:** Multiple layers using the ReLU activation function to capture intricate patterns.
* **Output Layer:** Uses the Sigmoid activation function to output probabilities, suited for binary classification.

The architecture's depth, controlled by the number of hidden layers, is crucial for learning from complex datasets.



**Model Implementation:**

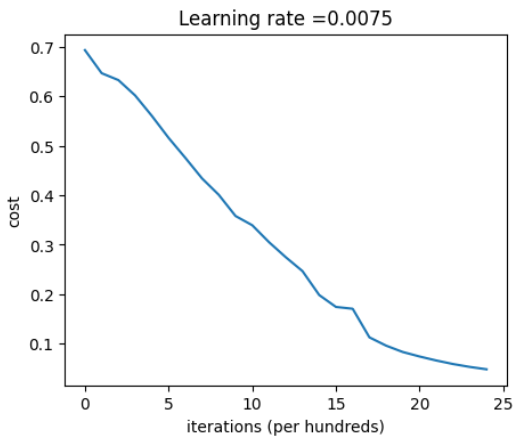
1. **Model Structure Definition:**
   * **Input Layer:** Accepts the feature vectors X
   * **Hidden Layers:**
     + These are crucial as they allow the network to learn non-linear representations.
     + The number of layers (depth) and the number of units in each layer (width) are hyperparameters.
   * **Output Layer:**
     + Utilizes the sigmoid activation function for binary classification, outputting probabilities.
2. **Parameter Initialization:**
   * **`initialize\_parameters` and `initialize\_parameters\_deep`:**
     + Random initialization of weights and biases is critical to prevent the network from being stuck in symmetric states.
     + For deep networks, initialization is done for each layer, ensuring that the variance of weights is controlled to prevent vanishing/exploding gradients.
3. **Forward Propagation:**
   * **linear\_forward:**
     + Computes the linear combination of inputs and weights for each layer.
     + Equation: 
   * **linear\_activation\_forward:**
     + Applies the activation function (ReLU for hidden layers and sigmoid for the output layer) to introduce non-linearity.
     + ReLU function: A= 
     + Sigmoid function: A= 
   * **L\_model\_forward:**
     + Implements the full forward propagation through all layers, combining linear and activation functions for the entire network.
     + The final output is the prediction Yhat​, which represents the model's classification probabilities.
4. **Cost Function Calculation:**
   * **compute\_cost:**
     + Uses the cross-entropy loss function, which is appropriate for binary classification tasks.
     + The loss guides the model on how well it's performing, with the goal of minimizing this value.
     + Equation: Cost



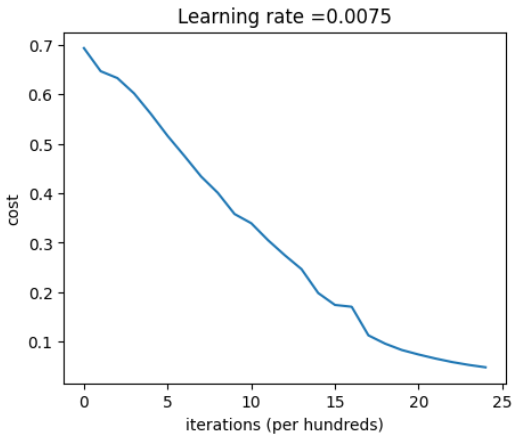
1. **Backward Propagation:**
   * **linear\_backward:**
     + Computes the gradients of the cost with respect to Z, W, and b for a single layer.
   * **linear\_activation\_backward:**
     + Combines the linear backward pass with the activation function's backward pass.
     + The gradients are propagated backward through the network, layer by layer, allowing the model to learn by updating parameters in the direction that reduces the cost.
   * **L\_model\_backward:**
     + Executes the backward propagation for the entire model, computing the gradients for all layers.
     + The chain rule of calculus is applied to efficiently compute these gradients, ensuring that each parameter is adjusted appropriately.
2. **Parameter Update:**
   * **update\_parameters:**
     + Updates the parameters (weights and biases) using gradient descent.
     + The learning rate determines the size of the steps taken towards minimizing the cost function.
     + Parameters are iteratively updated across all layers, gradually improving the model's performance.
3. **Model Training:**
   * **nn\_model:**
     + Integrates all the above functions into a single model that can be trained on the dataset.
     + It iteratively performs forward propagation, cost computation, backward propagation, and parameter updating for a specified number of epochs.
   * The model's predictions are compared against actual labels to evaluate performance.
4. **Prediction and Accuracy Evaluation:**
   * **predict:**
     + The final trained model is used to make predictions on new data.
     + The output is converted into binary labels (0 or 1) by thresholding the probabilities.
   * The model's accuracy is then calculated by comparing the predictions against the ground truth labels.

**Experiment Outputs:**

* **Accuracy:** The deep neural network achieved an accuracy of 72% on the test set using the `two\_layer\_model` and an accuracy of 80% on the test set using the `L\_layer\_model’
* **Learning Curve:** A plot of the cost function over iterations demonstrated the model's learning process, indicating convergence.
  + Two\_Layer\_Model:



* + L\_Layer\_Model:



**Results and Discussion:**

The deep neural network is trained on a given dataset, and its performance is evaluated using accuracy and loss metrics on the test data. The model’s ability to classify images accurately is analyzed, highlighting the importance of depth (i.e., number of layers) in capturing complex patterns.

By utilizing both ReLU and Sigmoid activations, the network effectively learns non-linear decision boundaries, which are critical for high performance in image classification tasks.

**GitHub Link:**