**Rashtreeya Shikshana Samithi Trust**

****

**School of Computer Science and Engineering**

## Bengaluru – 560059

CS3232 FUNDAMENTALS OF DEEP LEARNING

**V SEMESTER**

**B.Tech Computer Science & Engineering**

***LABORATORY RECORD***

**by**

**Francis Maria Sharan**

**USN: 1RVU22CSE055**

**2024-2025**

### LIST OF PROGRAMS

|  |  |
| --- | --- |
| **Sl. No.** | **Program** |
| 1. | Practice Python and NumPy basics |
| 2. | Implement logistic regression with a neural network mindset |
| 3. | Classify the planar dataset using one hidden layer |
| 4. | Build and use a deep neural network to classify a given image as cat or not a cat |
| 5. | MNIST dataset handwritten digit recognition using CNN |
| 6. | Classify flower image dataset using transfer learning of a pre-trained MobileNetV2 model |
| 7. | Use simple, deep, CNN-based denoising encoders on MNIST handwritten digit dataset. |
| 8. | Generate handwritten digit images using a deep convolutional generative adversarial network (DCGAN) and MNIST handwritten digit dataset. |
| 9. | Implement a character-level language model for name generation using recurrent neural networks. |
| 10. | Deploy multi-class (machine learning) classification of Iris dataset on AWS. |
|  |  |
|  |  |

|  |  |
| --- | --- |
| **Ex No: 1**  **Date: 07/08/2024** | **Python and NumPy basics** |

**Objective:**

To implement foundational functions for logistic regression, including the sigmoid function, vectorized operations, and performance optimization using Python and Numpy, as well as to measure computation time for improved efficiency in model training.

**Descriptions:**

In this lab, we explore key components and operations critical to logistic regression, a fundamental algorithm used for binary classification tasks. Logistic regression assigns probabilities to class labels based on input features and is widely used in applications such as image classification, fraud detection, and medical diagnosis.

Key concepts covered in this lab include:

* **Sigmoid Function**: The sigmoid function, often represented as  is used to map real-valued inputs to a range between 0 and 1, making it ideal for probability estimation in binary classification problems.
* **Vectorized Operations**: Efficient computation with numpy vectorized operations enables faster processing of large datasets by applying operations simultaneously across array elements, improving the model’s computational performance.
* **Computation Time Measurement**: Tracking execution time for operations helps optimize the logistic regression model by minimizing unnecessary computation overhead, a crucial aspect for scaling machine learning algorithms.

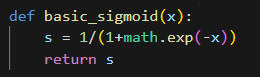
**Model:**

The logistic regression model in this lab is based on the following key steps:

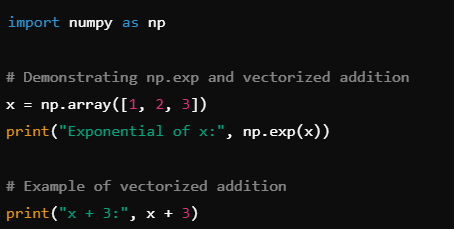
1. **Sigmoid Function**: The sigmoid function  is applied to map predictions to probabilities. This helps in determining the class likelihood for binary classification (1 for the positive class, 0 for the negative class).
2. **Vectorization for Efficiency**: Operations such as addition, subtraction, and dot products are vectorized using numpy. This allows the model to handle data in bulk, reducing computational load and increasing speed.
3. **Performance Optimization with Timing**: Using Python's time module, we measure the execution time for matrix and vector operations. Efficient use of vectorization and optimization techniques can significantly reduce the model's training time, which is essential for practical applications of logistic regression.

**Code Implementation**

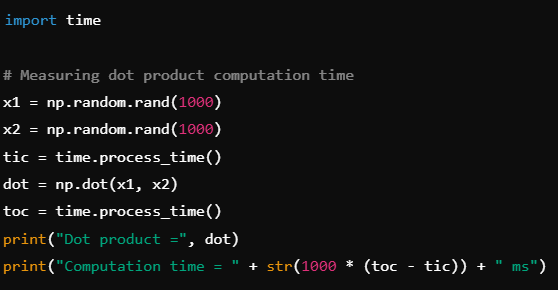
* Sigmoid Function Implementation:



* Vectorized Operations:



* Dot Product and Computation Time



**Results and Observations:**

Through this lab, we observe:

1. The **sigmoid function** successfully converts values to a probability range between 0 and 1, making it suitable for binary classification outputs.
2. **Vectorized operations** such as addition and dot products greatly improve the performance of basic operations, which is crucial in handling large datasets efficiently.
3. **Computation time measurements** show that vectorized numpy operations can process large arrays in milliseconds, underlining the importance of vectorization for scaling machine learning algorithms.

**Code:**

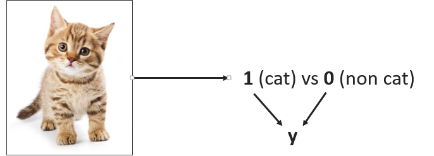
|  |  |
| --- | --- |
| **Ex No: 2**  **Date: 14/08/2024** | **Logistic Regression implementation** |

**Objective:**

To build a logistic regression classifier to recognize cat’s vs non cat using Gradient descent implementation.

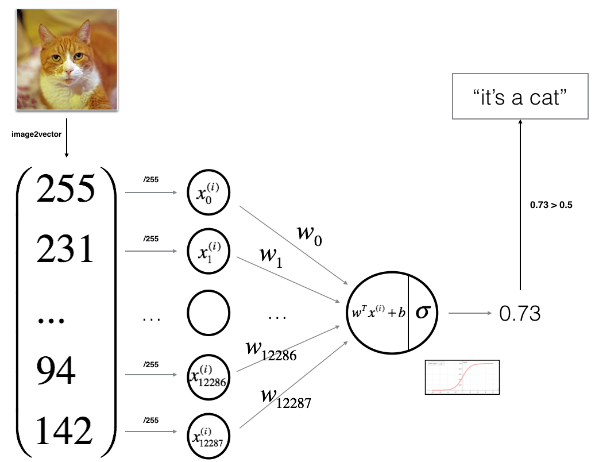
**Descriptions:**

Binary classification is the task of classifying elements of a given set into two groups. Logistic regression is an algorithm for binary classification. We have an input image x and the output y is a label to recognize the image. 1 means cat is on an image, 0 means that a non-cat object is in an image**.**



Logistic regression is a supervised learning algorithm that we can use when labels are either 0 or 1 and this is the so-called Binary Classification Problem. An input feature vector X may correspond to an image that we want to recognize as either a cat picture (1) or a non-cat picture (0). That is, we want an algorithm to output the prediction which is an estimate of y: Logistic Regression doesn't have a hidden layer. If you initialize the weights to zeros, the first example x fed in the logistic regression will output zero but the derivatives of the Logistic Regression depend on the input x (because there's no hidden layer) which is not zero. So at the second iteration, the weights values follow x's distribution and are different from each other if x is not a constant vector.

**Model:**

****

**Building the parts of algorithm**

The main steps for building a Neural Network are:

1. Define the model structure (such as number of input features)
2. Initialize the model's parameters
3. Loop:
   * Calculate current loss (forward propagation)
   * Calculate current gradient (backward propagation)
   * Update parameters (gradient descent)

**Code:**

|  |  |
| --- | --- |
| **Ex No: 3**  **Date:** | **Planar Data Classification using a Shallow Neural Network** |

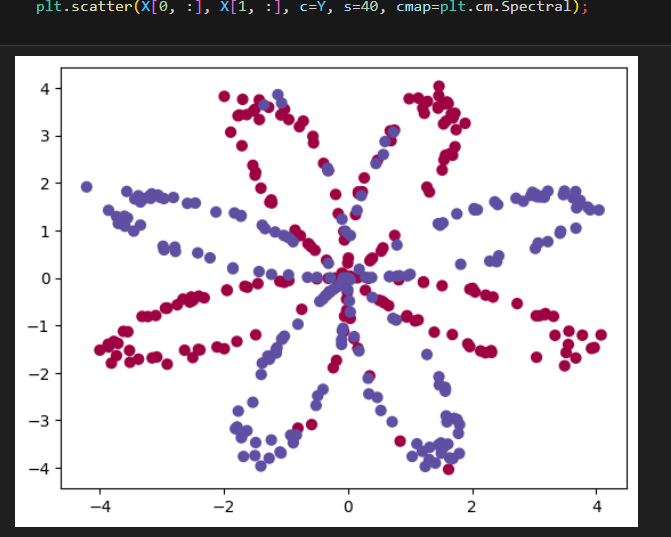
**Objective:**

To build a binary class classification neural network with a single hidden layer which uses activation functions tanh and sigmoid respectively. The Loss function used is the Cross-entropy loss.

**Descriptions:**

**Introduction to Binary Classification**

Binary classification involves categorizing data points into one of two classes. In this context, we will classify a "flower" dataset, which contains data points that belong to one of two classes. The data points are described by features X, and the output is represented by Y, where Y=0 or Y=1.

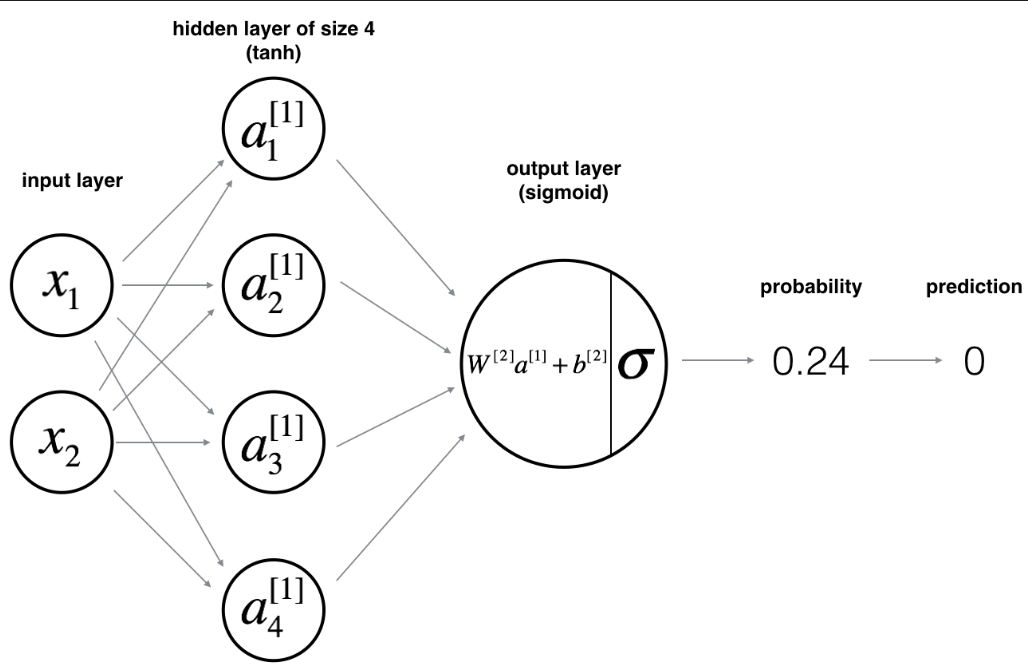


**Neural Network Architecture**

In this experiment, we design a neural network with one hidden layer. This hidden layer allows the network to capture more complex patterns in the data compared to logistic regression, which lacks a hidden layer.

* **Input Layer:** Receives the features XXX.
* **Hidden Layer:** Applies a non-linear activation function (tanh) to capture complex relationships.
* **Output Layer:** Uses the sigmoid function to predict probabilities for each class.

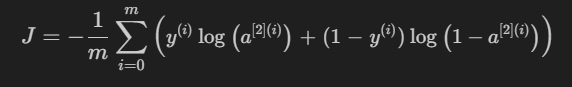
The model parameters (weights and biases) are initialized randomly and updated during training using gradient descent.



**Model Implementation**

The main steps for building this neural network are outlined below:

1. **Model Structure Definition**
   * Number of input features: The dimension of the feature vector X.
   * Number of hidden units: A hyperparameter that can be tuned for better performance.
   * Activation functions: We use tanh for the hidden layer and sigmoid for the output layer.
2. **Parameter Initialization**
   * The weights and biases are initialized randomly.
   * Initialization is crucial for ensuring that the network learns effectively during training.
3. **Forward Propagation**
   * Compute the linear combination of inputs and weights.
   * Apply the activation functions to introduce non-linearity.
   * Calculate the predicted output.
4. **Loss Calculation**
   * The cross-entropy loss function measures the discrepancy between the predicted and actual labels.
   * This loss guides the model on how to adjust its parameters.



1. **Backward Propagation**
   * Calculate the gradients of the loss function with respect to the parameters.
   * Use these gradients to update the parameters in the direction that reduces the loss.
2. **Parameter Update**
   * Gradient descent is employed to minimize the loss function, iteratively improving the model's performance.
3. **Building a Network using “*nn\_model()*”**
   * Integrating all the defined functions, namely forward propagation, cost computation, backward propagation, and parameter updating, into a model, thus adjusting the parameters and making predictions.

**Results and Discussion**

The experiment involves training the neural network on a planar dataset. The model's performance is evaluated by visualizing the decision boundary and comparing the predicted outputs with the actual labels. A well-trained model should accurately classify the data points, demonstrating the effectiveness of using a hidden layer for binary classification.

The developed shallow Neural Network resulted in an accuracy of 91%

**Code:**

|  |  |
| --- | --- |
| **Ex No: 4**  **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:**

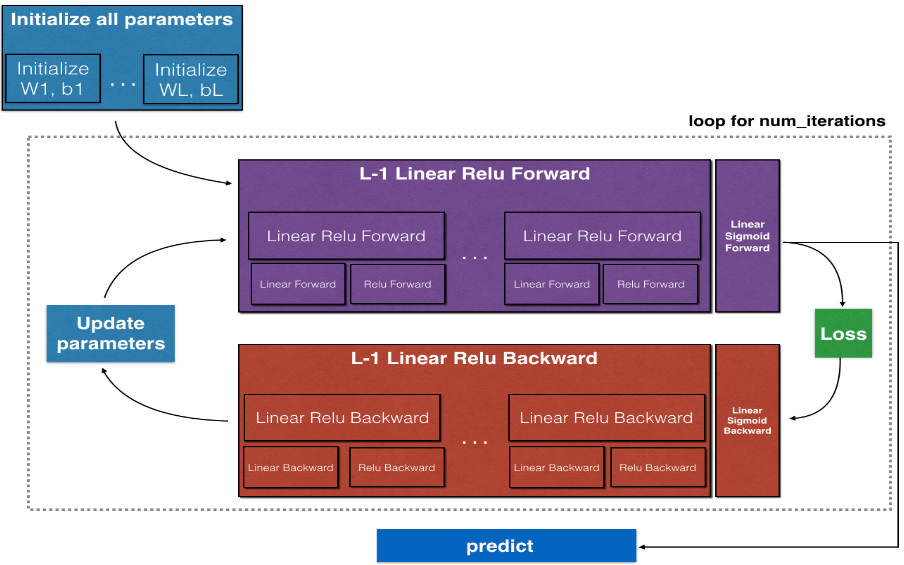
Deep learning involves training neural networks with multiple hidden layers to capture complex patterns in data. Image classification is a common application where a model is trained to recognize objects, animals, or scenes within images. This experiment demonstrates building a deep neural network using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

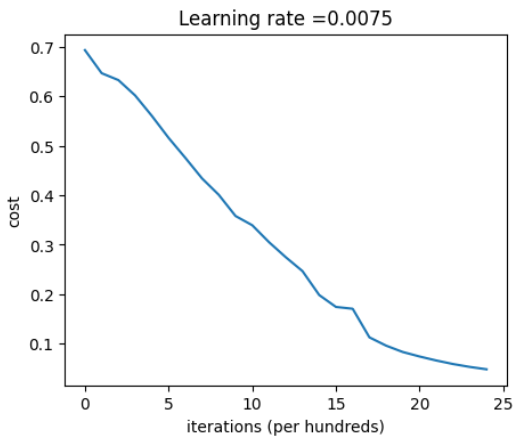
1. **Model Structure Definition:**
   * **Input Layer:** Accepts the feature vectors X
   * **Hidden Layers:**
     + These are crucial as they allow the network to learn non-linear representations.
     + The number of layers (depth) and the number of units in each layer (width) are hyperparameters.
   * **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



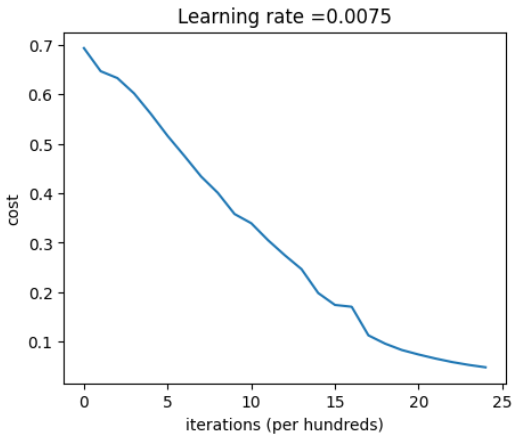
1. **Backward Propagation:**
   * **linear\_backward:**
     + Computes the gradients of the cost with respect to Z, W, and b for a single layer.
   * **linear\_activation\_backward:**
     + Combines the linear backward pass with the activation function's backward pass.
     + The gradients are propagated 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.

**Code:**

|  |  |
| --- | --- |
| **Ex No: 3**  **Date:20/08/2024** | **Building a Deep Neural Network** |

**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.

**Descriptions:**

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

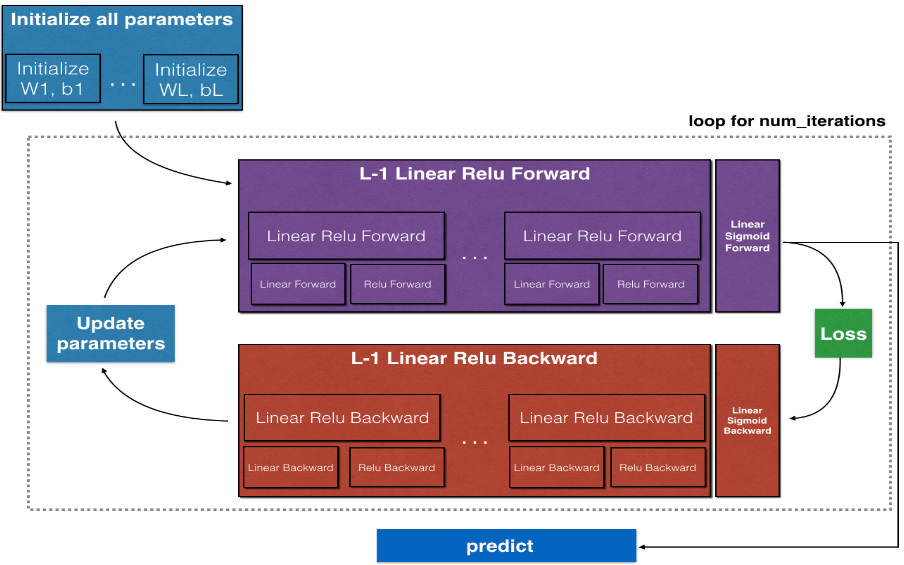
Deep learning involves training neural networks with multiple hidden layers to capture complex patterns in data. Image classification is a common application where a model is trained to recognize objects, animals, or scenes within images. This experiment demonstrates building a deep neural network using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

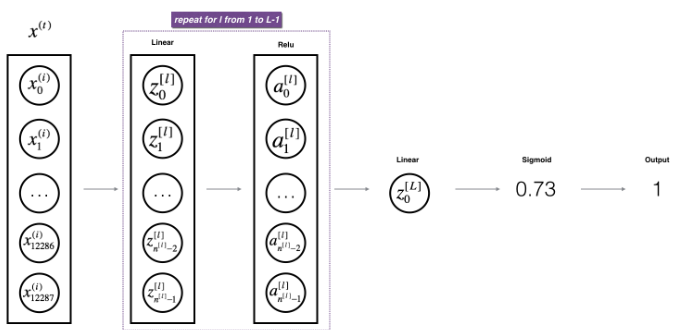
* Data is pre-processed, including normalization and splitting into training and test sets.
* Data augmentation may be applied to improve generalization.

**2. Initialization:**

* Parameters (weights and biases) are initialized using methods like Xavier/He initialization to facilitate better convergence during training.

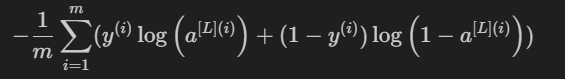
**3. Forward Propagation:**

* **Linear Step:** Each layer computes a linear combination of inputs and weights.
* **Activation Step:** ReLU is applied to hidden layers, and Sigmoid is applied to the output layer.



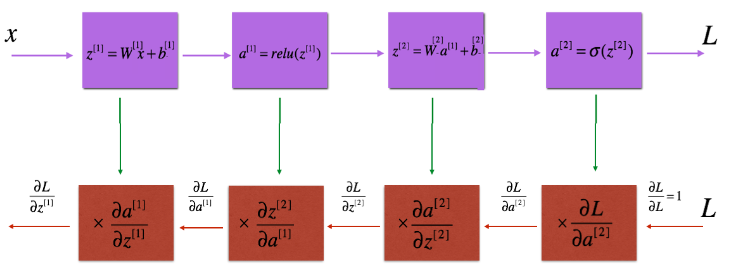
**4. Cost Function:**

* The cross-entropy loss function is utilized to quantify the error between predicted outputs and actual labels.



**5. Backward Propagation:**

* Gradients are computed with respect to each parameter using the chain rule, backpropagating from the output to the input layer.



**6. Parameter Update:**

* Parameters are updated using the computed gradients, typically employing optimization algorithms like gradient descent.

**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.

**Code:**

|  |  |
| --- | --- |
| **Ex No: 4**  **Date: 28/08/24** | **Convolutional Neural Network (CNN) Implementation for Handwritten Digit Recognition** |

**Title:**Convolutional Neural Network (CNN) Implementation for Handwritten Digit Recognition

**Objective:**To develop and implement a Convolutional Neural Network (CNN) model to recognize handwritten digits from the MNIST dataset using Python and TensorFlow.

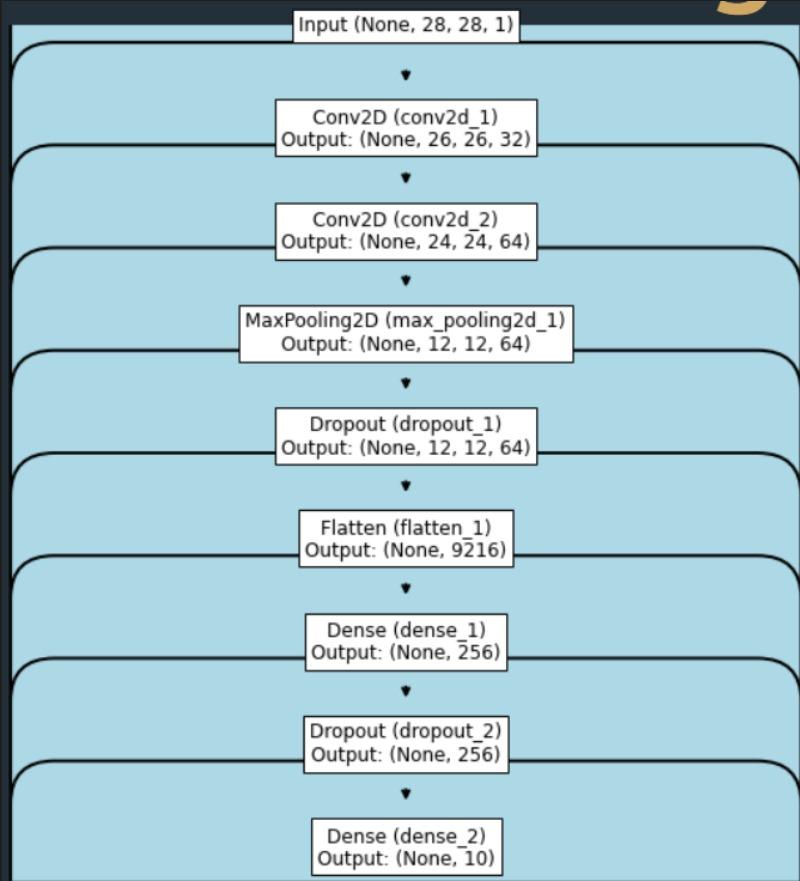
**Description:**Handwritten digit recognition is a classical problem in computer vision and pattern recognition. The MNIST dataset is a well-known dataset comprising 70,000 images of handwritten digits (0-9) in grayscale, with each image having a resolution of 28x28 pixels.

Convolutional Neural Networks (CNNs) are a class of deep neural networks that have proven highly effective for image classification tasks. A CNN typically consists of convolutional layers, pooling layers, and fully connected layers. The convolutional layers act as feature extractors, while the fully connected layers map these extracted features to the final output classes.

In this lab, we implement a CNN model using TensorFlow and Keras to classify images from the MNIST dataset. The model architecture includes convolutional layers followed by max-pooling layers, dropout for regularization, and dense layers for classification. The model is trained and evaluated on the MNIST dataset.

**Model Architecture:**

1. **Input Layer:** 28x28 grayscale images.
2. **Convolutional Layer 1:** 32 filters, 3x3 kernel size, ReLU activation.
3. **Max Pooling Layer 1:** 2x2 pool size.
4. **Convolutional Layer 2:** 64 filters, 3x3 kernel size, ReLU activation.
5. **Max Pooling Layer 2:** 2x2 pool size.
6. **Flatten Layer:** Flattening the 2D matrix into a 1D vector.
7. **Dense Layer 1:** 128 units, ReLU activation.
8. **Dropout Layer:** 0.5 dropout rate.
9. **Output Layer:** 10 units (corresponding to the 10 classes), softmax activation.



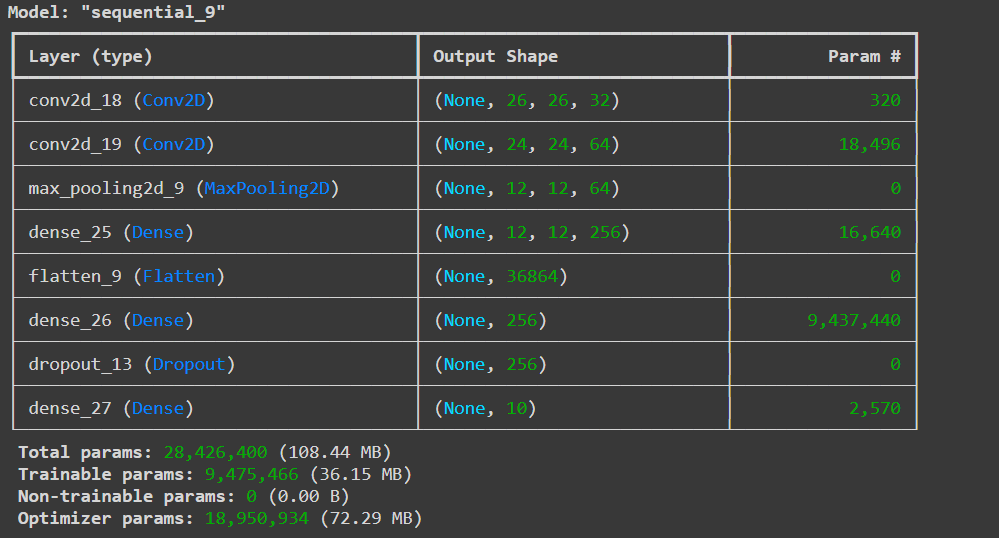
**Procedure:**

1. **Data Preparation:**
   * Import the MNIST dataset from TensorFlow/Keras.
   * Preprocess the data by normalizing pixel values to the range [0, 1].
   * Split the data into training and testing sets.
2. **Model Building:**
   * Define the CNN architecture as outlined above.
   * Compile the model with appropriate loss function (categorical\_crossentropy), optimizer (Adam), and metrics (accuracy).
3. **Training:**
   * Train the model on the training data with a specified number of epochs and batch size.
   * Monitor the training process and validation accuracy.
4. **Evaluation:**
   * Evaluate the model's performance on the test set.
   * Plot the training and validation accuracy/loss curves.
5. **Prediction:**
   * Use the trained model to predict classes for new input images.
   * Display a few test images along with their predicted labels.
6. **Hyperparameter Tuning:**

* Tuned the batch size by reducing it from 128 to 100
* Increased the epochs to 15
* Added another dense layer of 256 neurons with ‘ReLU’ Activation function
* Removed the initial dropout layer and changed the second one to 0.25

**Results:**

* The CNN model achieved an accuracy of 92.4% on the test dataset.
* Training and validation accuracy/loss curves indicates the model is optimal for the given problem statement.

****

**Conclusion:**

The implementation of a CNN model for handwritten digit recognition using TensorFlow was successful. The model was able to classify digits with high accuracy, demonstrating the effectiveness of CNNs in image classification tasks. Further improvements could include experimenting with different architectures or hyperparameters to optimize performance.

**Code:**

|  |  |
| --- | --- |
| **Ex No: 5**  **Date: 04/9/24** | **CNN Transfer Learning** |

**Title**: Transfer Learning using CNN for Image Classification

**Objective**:  
To implement transfer learning using a pre-trained Convolutional Neural Network (CNN) model for image classification using TensorFlow and Keras.

**Description**:  
Transfer learning is a machine learning technique where a pre-trained model is reused for a different but related task. It is particularly useful when a large dataset is unavailable for training from scratch. In this lab, we use a CNN model pre-trained on the ImageNet dataset and apply it to classify images from a new dataset. We will fine-tune the model's top layers and use the pre-trained layers to extract useful features.

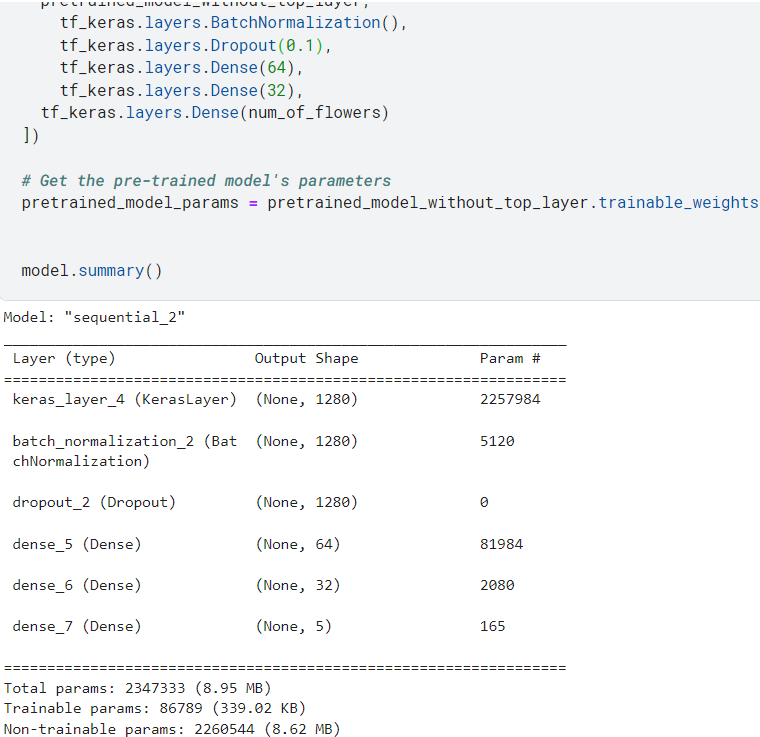
**Model Architecture**:

* **Pre-Trained Model**: Use a pre-trained model (e.g., VGG16, ResNet50) without the top fully connected layers.
* **Flatten Layer**: Flatten the extracted features into a 1D vector.
* **Dense Layer 1**: 512 units with ReLU activation.
* **Dropout Layer**: 0.5 dropout rate for regularization.
* **Output Layer**: A dense layer with softmax activation corresponding to the number of classes in the new dataset.

**Procedure**:

1. **Data Preparation**:
   * Load the new dataset.
   * Preprocess the images (resize to the input shape required by the pre-trained model, normalize pixel values to the range [0, 1]).
   * Split the dataset into training and validation sets.
2. **Model Building**:
   * Load a pre-trained model (e.g., ResNet50) from Keras applications, without the top layers.
   * Freeze the layers of the pre-trained model to retain the learned features.
   * Add new layers on top for fine-tuning, as described in the model architecture.
3. **Model Compilation**:
   * Compile the model using categorical\_crossentropy as the loss function, Adam optimizer, and accuracy as the metric.
4. **Training**:
   * Train the model on the new dataset using a specified number of epochs and batch size.
   * Monitor the training process using validation accuracy.
5. **Evaluation**:
   * Evaluate the model's performance on the validation/test set.
   * Plot the training and validation accuracy/loss curves.
6. **Prediction**:
   * Use the trained model to predict classes for new input images.
   * Display a few test images with their predicted labels.
7. **Fine-Tuning**:
   * Fine-tuning involves taking a pre-trained model, such as MobileNet V2, and adapting it to a new classification task. Initially, the pre-trained layers are frozen to retain the features learned from a large dataset like ImageNet. After adding new layers specific to the task, selected pre-trained layers are gradually unfrozen and retrained on the new dataset. This allows the model to adjust its features while still leveraging the knowledge gained from its previous training. We added a BatchNormalization and a Dropout layer to reduce overfitting and then added two dense layers of 64 and 32 neurons to get better fitting. We tuned the batch size, epochs and validation data size to get the best accuracy and lowest loss possible.

**Results**: The model achieved an accuracy of 86.05% and loss of 44.3% on the validation dataset.





**Conclusion**: Transfer learning with a pre-trained CNN model was successfully implemented for image classification. Fine-tuning the model improved its performance, demonstrating the effectiveness of using pre-trained models for similar image classification tasks. Further improvements could include adjusting the learning rate or exploring different pre-trained models.

**Code:**

|  |  |
| --- | --- |
| **Ex No: 6**  **Date: 09/09/24** | **Autoencoder using the MNIST dataset** |

**Objective:**

To build an autoencoder that compresses MNIST images into a lower-dimensional representation and then reconstructs the original images from this compressed format.

**Descriptions:**

In this exercise, we broke the implementation down into three steps:

**Data Preparation:** MNIST images are loaded, normalized, and flattened into 784-dimensional vectors using TFDS.

**Model Architecture:**

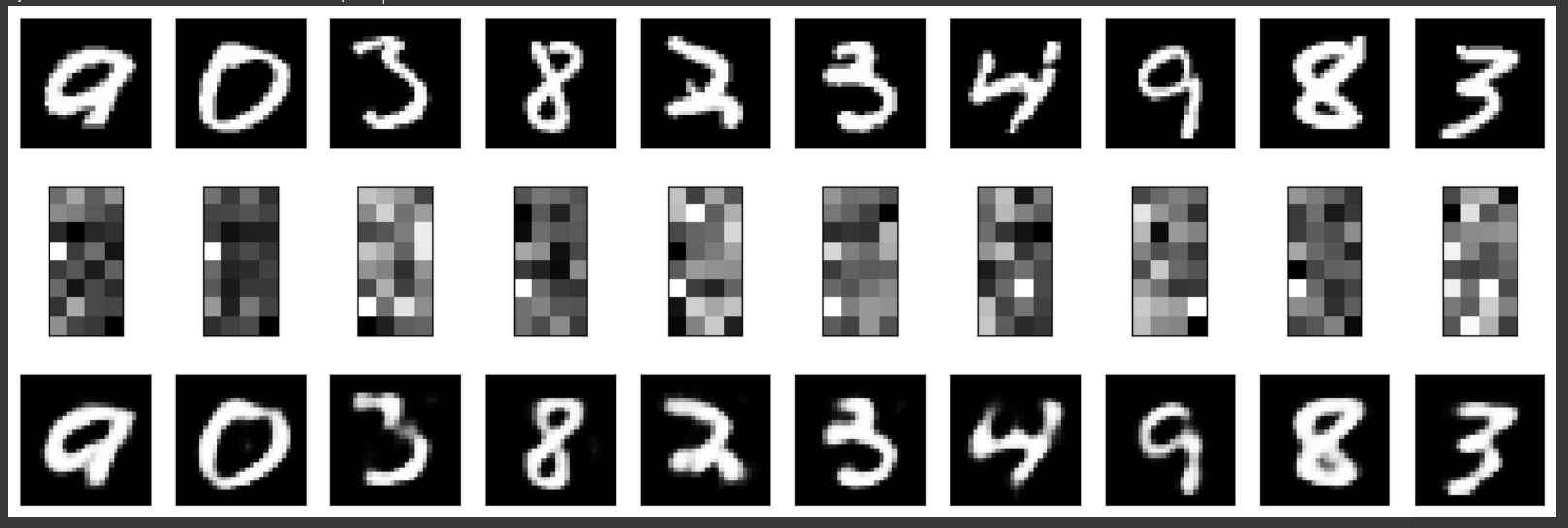
* **Encoder:** A dense layer with 32 units (ReLU) compresses the input.
* **Decoder:** A dense layer with 784 units (sigmoid) reconstructs the images.

**Training:** The model is compiled with the Adam optimizer and binary cross-entropy loss, then trained for 50 epochs.

**Model:**

* **Encoder**: A dense layer with 32 units and ReLU activation compresses the input image to a 32-dimensional representation.
* **Decoder**: A dense layer with 784 units and sigmoid activation reconstructs the original 784-dimensional image from the compressed representation.

**Result and Analysis:**



**Code: Add codes of 6.1, 6.2 and 6.3**

|  |  |
| --- | --- |
| **Ex No: 7**  **Date: 11/09/24** | **Autoencoder using the Fashion MNIST Dataset**  **(With Denoising the dataset)** |

**Objective:**

To build an autoencoder that denoises Fashion MNIST images by compressing them into a lower-dimensional representation and reconstructing the original images from the compressed format.

**Descriptions:**

In this exercise, we broke the implementation down into three steps:

**Data Preparation:**

* Fashion MNIST images are loaded and normalized using TensorFlow Datasets (TFDS).
* Noise is added to the images, and the noisy images are used as inputs to the model, while the clean images are used as targets.

**Model Architecture:**

* **Encoder:** Consists of two Conv2D layers followed by MaxPooling layers, reducing the image size while extracting features.
* **Bottleneck:** A dense representation compresses the information further.
* **Decoder:** Upsamples the compressed representation back to the original image size using Conv2D and UpSampling2D layers.

**Training:**

* The model is compiled with the Adam optimizer and binary cross-entropy loss.
* The training is performed for 40 epochs with a batch size 128.

**Model:**

**Encoder:**

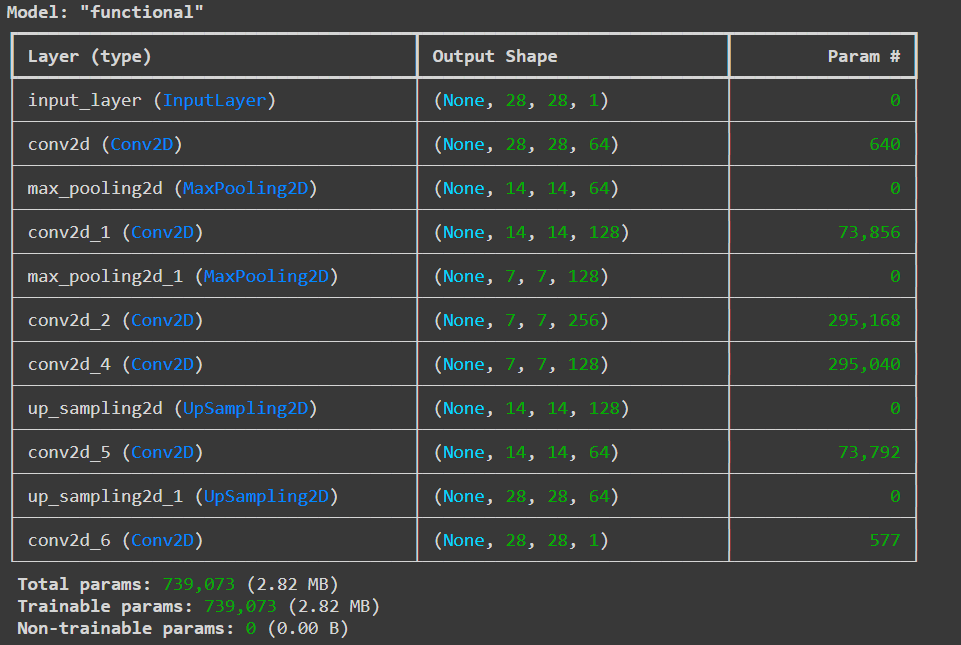
* Two convolutional layers with 64 and 128 filters respectively, each followed by a MaxPooling layer.
* Compresses the image down to a smaller representation.

**Bottleneck:**

* A convolutional layer with 256 filters that acts as the compressed representation layer.

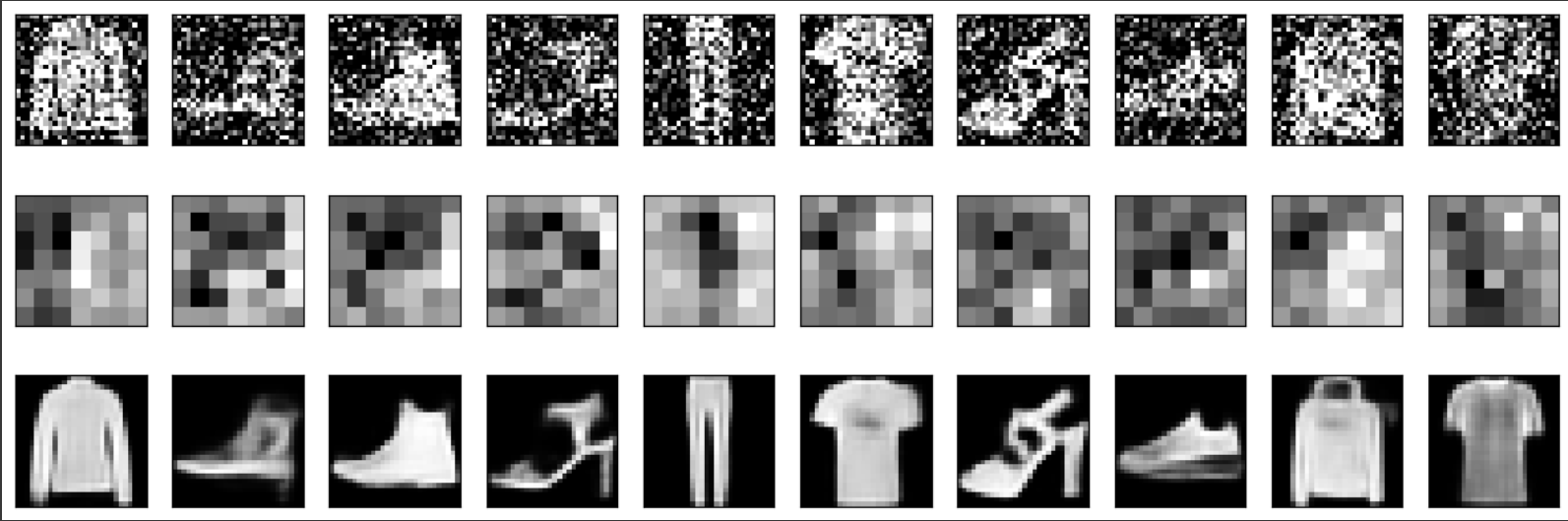
**Decoder:**

* Two convolutional layers that upsample the encoded representation back to the original image size.



**Result and Analysis:**

The autoencoder learned to denoise the images by reconstructing them from noisy inputs. The clean outputs show that the model effectively removed noise, retaining key details of the original images.



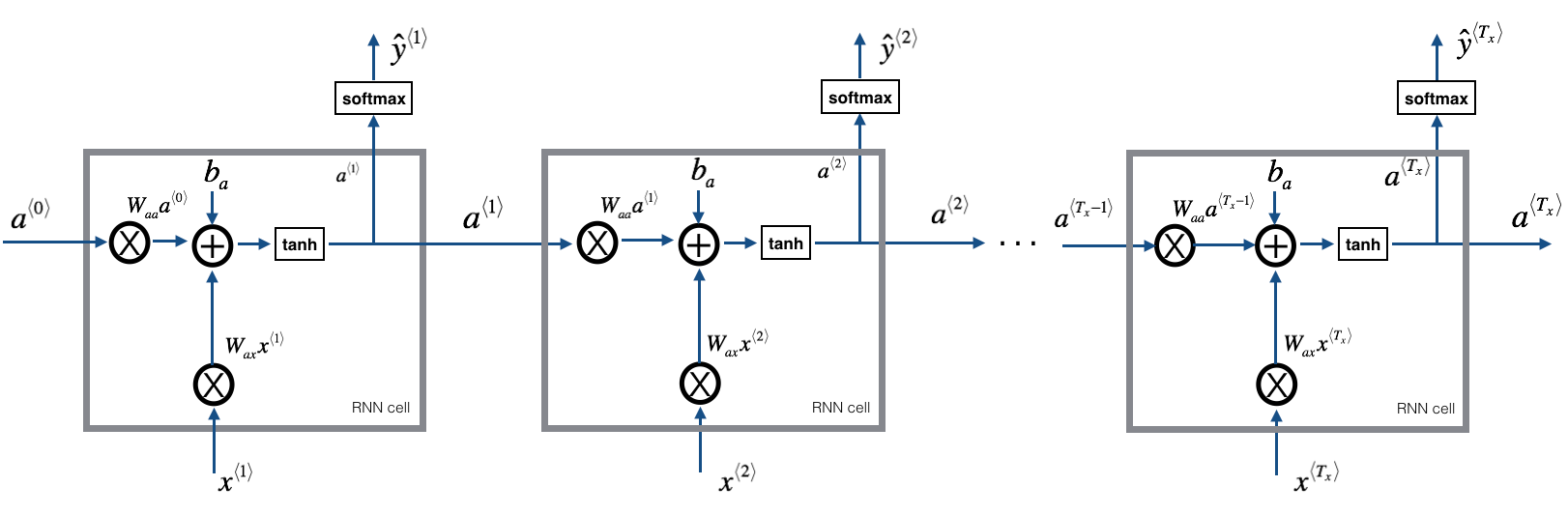
**Code:**

|  |  |
| --- | --- |
| **Ex No: 8**  **Date: 25/09/24** | **RNN** |

**Objective:**

The objective of this lab is to understand and implement a Recurrent Neural Network (RNN) using Python and a suitable deep learning library (e.g., TensorFlow or PyTorch). The focus is on sequence modelling tasks such as time series prediction or text processing.

**Description:**

****

This lab involves the step-by-step implementation of an RNN. Key processes include data preprocessing, model building, training, and evaluation. The code explanations will cover these steps in detail.

**Code Explanation**

1. **Importing Libraries**

Import necessary libraries for data manipulation (numpy, pandas), plotting (matplotlib), and deep learning (keras).

1. **Loading and Preprocessing Data**
   * **Purpose:** Load data from a CSV file, select the relevant column, and scale the data to a range of 0 to 1 using Min-Max Scaling. This is crucial for efficient model training.
   * **Expected Output:** The data will be transformed into scaled values, usually between 0 and 1.
2. **Creating Training and Testing Datasets**
   * **Purpose:** Split the scaled data into training (80%) and testing (20%) datasets to evaluate the model's performance.
   * **Expected Output:** Two datasets: train for model training and test for evaluation.

**4. Converting Data to RNN-Compatible Format**

* + **Purpose:** Transform the data into sequences of a specified time step to make it compatible with RNN input requirements.
  + **Expected Output:** X\_train, y\_train, X\_test, and y\_test arrays containing input features and targets for training and testing.

1. **Reshaping Input for RNN**

* **Purpose:** Reshape the input data into a three-dimensional format expected by RNNs: [samples, time steps, features].
* **Expected Output:** Reshaped data suitable for feeding into an RNN.

1. **Building the RNN Model**

* **Purpose:** Define an RNN model using a Sequential architecture with one Simple RNN layer of 50 units and a dense layer for output.
* **Expected Output:** Compiled RNN model ready for training.

1. **Training the Model**

* **Purpose:** Train the RNN model on the training data over 50 epochs with a batch size of 32, using validation data to monitor performance.
* **Expected Output:** Training process with loss values per epoch, showing improvement over time.

**8. Model Evaluation and Predictions**

* **Purpose:** Generate predictions on both training and testing data. Predictions are then inverse scaled back to their original range for evaluation.
* **Expected Output:** Predictions for both training and testing data in the original data scale.

**Key Processes and Main Observations:**

* **Data Preparation:** Loading and scaling of data to normalize input values.
* **Sequence Creation:** Transformation of data into time-step sequences for RNN compatibility.
* **Model Building:** SimpleRNN model with one hidden layer and an output layer, optimized using Adam.
* **Training and Evaluation:** Continuous training and validation using Mean Squared Error to minimize prediction errors.

**Code:**