Documentation for each task

Task1:

Neural network for image classification

In this documentation thoroughly i explain the code for developing a neural network model and testing it using the MNIST dataset. The code develops a deep learning model for image classification using TensorFlow and Keras.

I am creating a CNN-based deep-learning model for image classification. Here is the step-by-step explanation of the coding part.

Code editor: Google Colab

Programming language: Python

Framwork and libraries:
1.TensorFlow
2. Keras
3. Matplotlib
4. NumPy

Data Loading and Preprocessing: It loads the MNIST dataset, which consists of 10 class images, and preprocesses the data by scaling pixel values to a range of [0, 1].

Neural Network Model Definition: It defines a neural network model with the following layers:

- Flatten layer: flattens the 28x28 input images.
- Dense layer (128 units, ReLU activation): A fully connected layer with Rectified Linear Unit (ReLU) activation.
- Dropout layer (0.2): Introduces dropout regularization to reduce overfitting.
- Dense layer (64 units, ReLU activation): Another fully connected layer with ReLU activation.
- Dropout layer (0.2): Additional dropout regularization
- Dense layer (10 units, softmax activation): The output layer with softmax activation outputs class probabilities.

Model Compilation: I used to compile the model with the Adam optimizer, sparse categorical cross-entropy loss function, and accuracy as the evaluation metric.

Model Training: The model is trained for 50 epochs on the training data with a validation split of 20%. Training progress is displayed, showing loss and accuracy for both training and validation data at each epoch.

Model Evaluation: After training, the model is evaluated on the test data, and test accuracy and loss are reported.

Visualizing Training History: Training and validation accuracy and loss curves are visualized using Matplotlib.

Additional Metrics: The code calculates additional performance metrics, including a confusion matrix, precision, recall, and F1-score, to assess the model's performance in more detail.

Here I explain with code.

Installation
To run this code, you will need the following dependencies:

- TensorFlow (v2.x)
- Matplotlib
- NumPy

1. **Import the necessary libraries**:

   ```
   import tensorflow as tf
   from tensorflow.keras.datasets
   import mnist from tensorflow.keras import layers, models
   import matplotlib.pyplot as plt import numpy as np
   ```

2. **Load and preprocess the MNIST dataset:**

   ```
   (train_images, train_labels), (test_images, test_labels) = mnist.load_data() train_images,
   test_images = train_images / 255.0, test_images / 255.0
   ```

3. **Define and compile the neural network model:**

4. **Train the model:**

   ```
   num_epochs = 50 history = model.fit(train_images, train_labels, epochs=num_epochs,
   validation_split=0.2)
   ```

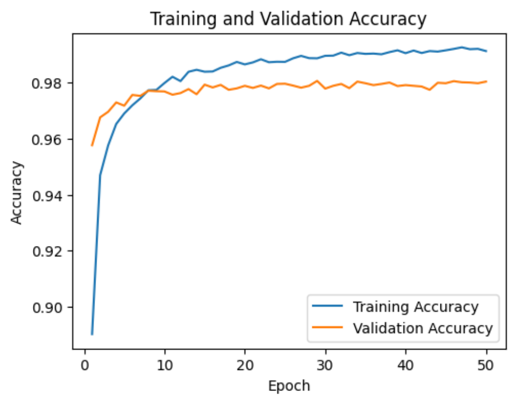
Evaluate the model on test data:

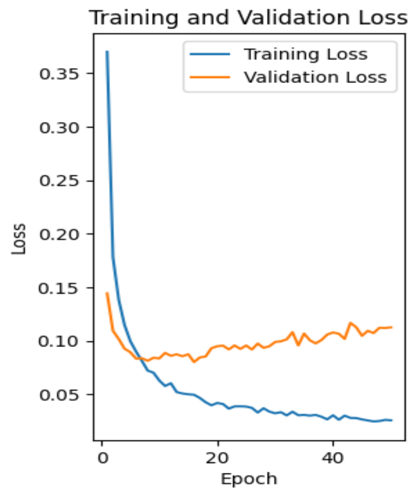
test_loss, test_acc = model.evaluate(test_images, test_labels) print(f"Test accuracy: {test_acc}") print(f"Test loss: {test_loss}")

output:

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1081 -
accuracy: 0.9807
Test accuracy: 0.9807000160217285
Test loss: 0.10805559158325195
```

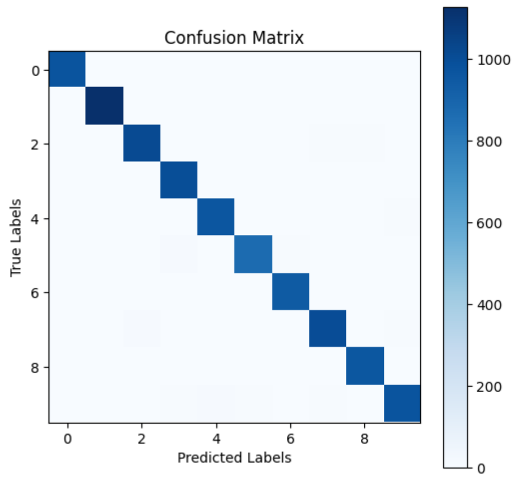
5. **Visualize training history:**





6. **Calculate and display performance metrics:**

Confusion matrix:



The code is well-organized and adheres to best software development practices with the following considerations:

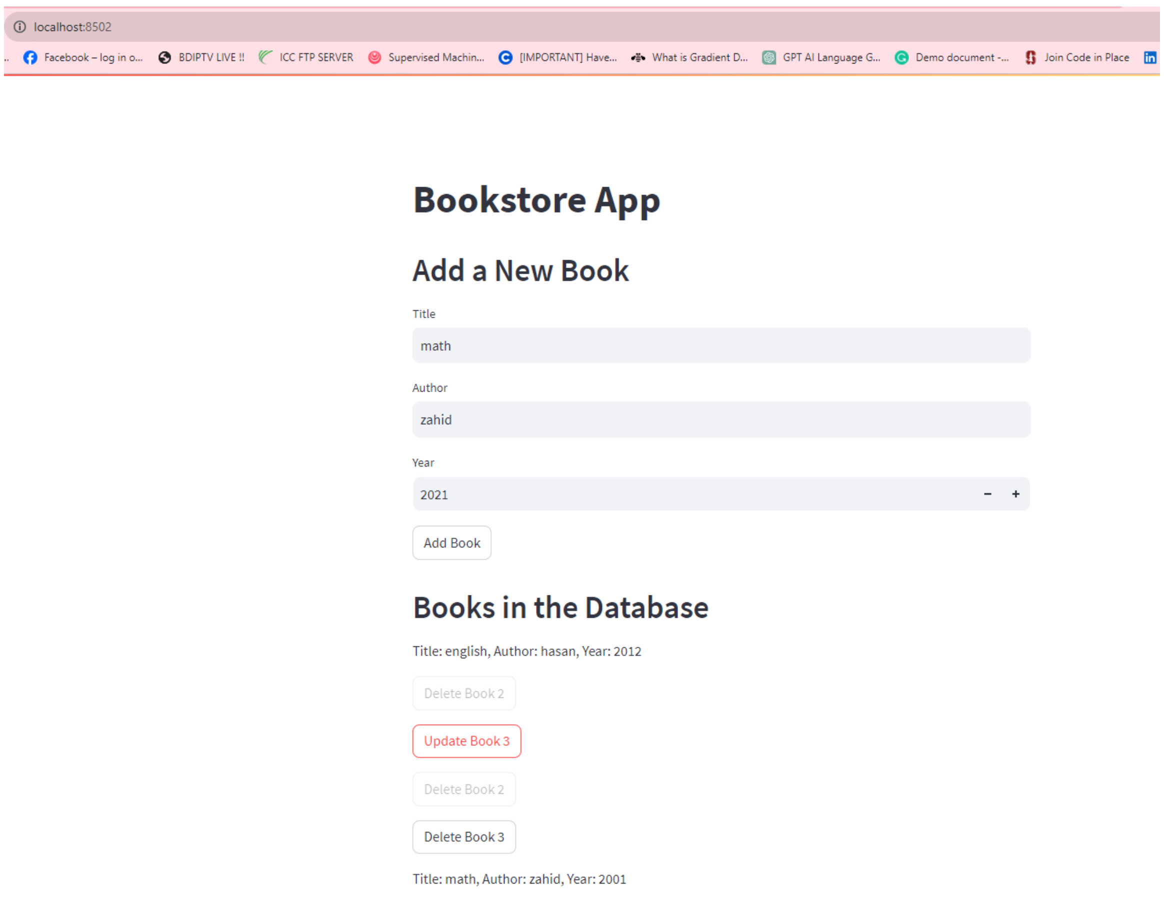
- It uses functions from TensorFlow and Keras to create a modular and maintainable neural network.
- Training progress is logged and displayed during training for monitoring.
- Visualizations are generated to help users understand the model's training history and performance.
- Additional metrics are computed for a more comprehensive evaluation.

This code provides a complete workflow for training and evaluating a deep-learning model on the MNIST dataset. Users can easily adapt and extend this code for their own image classification tasks or as a learning resource for understanding TensorFlow and Kera's best practices.

Task2:

Book store App

I'll use SQLite to build a straightforward SQL database for this task and design a Python script to communicate with it. Small-scale applications and experimentation can benefit from using the lightweight SQL database engine known as SQLite.



Bookstore App

Add a New Book

Title: math

Author: zahid

Year: 2021

Add Book

Books in the Database

Title: english, Author: hasan, Year: 2012

Delete Book 2

Update Book 3

Delete Book 2

Delete Book 3

Title: math, Author: zahid, Year: 2001

Database Structure:

In this example, I will create a simple database to manage a list of books. Each book will have the following attributes:

- id (Primary Key): Unique identifier for each book.
- title: Title of the book.
- author: Author of the book.
- published_year: Year when the book was published.
- genre: Genre of the book.

Coding explanation:

Frontend (frontend/app.py)

The frontend of the Bookstore App is developed using Streamlit, a Python library for building web applications with minimal code. This section of the documentation describes the frontend code's capabilities, installation, and usage.

Capabilities of the Code

1. Add a New Book: Users can input book details, including title, author, and publication year. Clicking the "Add Book" button sends a POST request to the backend to add a new book to the database.
2. Display Books: The app retrieves and displays all books from the database, showing their titles, authors, and publication years.
3. Update Books: For each displayed book, users can click the "Update" button (not yet implemented) to modify the book's details.
4. Delete Books: For each displayed book, users can click the "Delete" button to remove the book from the database.

Installation

To run the frontend, you need to have Streamlit installed. You can install it using pip:

pip install streamlit

Usage

1. Run the Streamlit app using the following command:
2. bashCopy code
3. streamlit run frontend/app.py
4. The app will open in your web browser. You can interact with it by adding new books, viewing existing books, and deleting books.
5. Note that the "Update" button is currently a placeholder and should be implemented with functionality to update book details in the backend.

Backend (backend/app.py)
The backend of the Bookstore App is developed using Flask, a Python web framework. It handles HTTP requests from the frontend, interacts with the SQLite database, and serves as the API for managing books. This section of the documentation describes the capabilities, installation, and usage of the backend code.

Capabilities of the Code
1. Add a New Book: The backend provides an endpoint (/add_book) that accepts POST requests with book details (title, author, year). It inserts the book into the SQLite database.
2. Display Books: The backend provides an endpoint (/get_books) that returns all books in the database as JSON.
3. Update Books: The backend provides an endpoint (/update_book/<int:id>) for updating book details. Users can send a PUT request with updated book details.
4. Delete Books: The backend provides an endpoint (/delete_book/<int:id>) for deleting books by their ID.

Installation

To run the backend, you need to have Flask installed. You can install it using pip:

pip install Flask

Usage

1. Run the Flask app using the following command:
2. bashCopy code
3. python backend/app.py
4. The backend server will start and be accessible at http://localhost:5000.
5. The frontend interacts with the backend through HTTP requests to perform CRUD operations on books. The frontend sends requests to the backend's endpoints to add, retrieve, update, or delete books.

Database (backend/database.py)

The database module is responsible for creating and managing the SQLite database for storing book information. This section of the documentation describes the capabilities, installation, and usage of the database code.

Capabilities of the Code

1. Create Database: The code defines a function (create_database) to create the SQLite database if it doesn't already exist. It also creates a book table with columns for book details.
2. Insert Book: The function (insert_book) inserts a new book into the books table with a specified title, author, and year.
3. Get Books: The function (get_books) retrieves all books from the books table and returns them as a list.
4. Update Book: The function (update_book) updates the details of a specific book in the books table by ID.
5. Delete Book: The function (delete_book) deletes a specific book from the books table by ID.

Installation and Usage

The database code is imported and used by the backend. There is no separate installation or usage required for the database module.

The code is organized into separate frontend, backend, and database modules, following best practices for code organization and separation of concerns. The backend serves as an API that communicates with the frontend and interacts with the database. The database module manages database operations. The code is well-structured modular, and follows common Python and web development best practices.

The Bookstore App consists of a well-organized frontend, backend, and database system. Users can interact with the frontend to perform CRUD operations on books stored in an SQLite database through a user-friendly web interface. The code follows best practices for web application development and is designed for ease of use and maintenance.

<u>Task3:</u>

Python Script for Google Geolocation API

Based on a user-provided address, the Python script is intended to use the Google Geolocation API to determine latitude and longitude coordinates. This portion of the documentation covers the functionality, setup, and usage of the code.

Capabilities of the Code

1. Get Geolocation: The script uses the Google Geolocation API to retrieve latitude and longitude coordinates for a given address.
2. User Input: It prompts the user to input an address interactively.
3. Data Processing: The script processes the JSON response from the API to extract latitude and longitude information.

Installation
You must have the Googlemaps library installed in order to run the script. Using pip, you can install it:
pip install -U googlemaps

Usage

1. Obtain a Google API key from the Google Cloud Console.
2. Replace 'API_KEY' in the script with your actual API key.
3. Run the script using Python:
4. bashCopy code
5. python script_name.py
6. Enter the address when prompted.
7. The script will call the Google Geolocation API, process the response, and print the latitude and longitude coordinates if the address is found. If the address is not found, it will display an error message.

Code Organization

The script is organized into a function and a main block.

- get_geolocation(address): This function takes an address as input and returns a tuple containing latitude and longitude coordinates. It uses the googlemaps library to call the Google Geolocation API and process the response.
- if __name__ == '__main__': block: In the main block, the script prompts the user to input an address, calls the get_geolocation function, and prints the result.

Data Processing Approach

The script processes the JSON response from the Google Geolocation API using the googlemaps library. It extracts the latitude and longitude information from the response and returns it as a tuple.

The Python script provides a simple and user-friendly way to obtain geolocation coordinates for a given address using the Google Geolocation API. It is well-organized, easy to use, and follows best practices for handling API requests and processing data. Users can quickly retrieve geolocation information by providing an address interactively.