CNN Design Assistant

Evan Lagan
21489316

Final Year Project – 2025
B.Sc. Single Honours in
Computer Science and Software Engineering

Department of Computer Science
Maynooth University
Maynooth, Co. Kildare
Ireland

A thesis submitted in partial fulfilment of the requirements for the B.Sc. Single/Double Honours in Computer Science/Computer Science and Software Engineering.

Supervisor: Charles Markham

Contents

List of Figures

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study as part of the Bachelor of Science qualification (Computer Science & Software Engineering), is *entirely* my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

I hereby acknowledge and accept that this thesis may be distributed to future final year students, as an example of the standard expected of final year projects.

Signed: Evan Lagan Date: 16/03/2025

Acknowledgements

I would like to thank my parents for their support.

Abstract

This report presents the development of a web-based application designed to make CNN model development more accessible. It aims to provide a more structured and user-friendly approach to CNN model creation and training, particularly for those with limited coding experience. The introduction of this report outlines the project's motivation and objectives, the technical background provides an overview of the technologies used in the applications development. The solution details the system's design and implementation, and the evaluation assesses its functionality. Finally, the conclusion summarizes the findings and discusses potential future improvements.

Chapter 1: Introduction

Convolutional Neural Networks (CNNs) are fundamental to modern deep learning applications and image recognition tasks, requiring careful design and solid understanding of underlying concepts for effective implementation. This project addresses the challenges of CNN design by presenting a web-based solution to assist both beginners and experts.

1.1 Motivation

Developing and understanding the functionality of Convolutional Neural Networks (CNNs) can be a time-consuming process, requiring both theoretical knowledge and practical coding experience. Traditional approaches of developing CNN models often involve extensive coding frameworks like TensorFlow or PyTorch, making it challenging for beginners to get started and for experienced users to rapidly prototype and test models [1][2]. This project is driven by the need for a tool that accelerates the learning process and facilitates experimentation without the immediate requirement for proficiency in any specific framework.

1.2 Statement of the problem

Despite the growing adoption of deep learning, CNN model development remains largely inaccessible to those without programming expertise. Beginners often struggle understanding model architectures and configuring hyperparameters, as most resources are scattered across articles and framework documentation. Meanwhile, experienced users frequently face inefficiencies when manually coding and debugging model architectures, leading to wasted time. Existing tools that offer visualization of model training typically require programming knowledge and others lack support for custom datasets, limiting their accessibility and usability [3][4]. This project seeks to bridge these gaps by providing a solution that enhances accessibility for beginners while improving efficiency for experienced users. By simplifying the model development process and reducing reliance on extensive coding, this tool will support a broader range of users in building and experimenting with CNN architectures.

1.3 Justification

Currently, no existing platforms offer both an intuitive model building interface and automatic Python code generation for CNN development. While tools like ChatGPT, Copilot, and other LLMs can help users learn fundamental concepts and generate working code, the depth of understanding gained depends on the user, and important aspects may be overlooked. Given the increasing demand for CNN skills across various fields a more accessible and comprehensive solution is highly valuable. For example, a biologist developing a model to classify different pollen types may struggle with coding-heavy resources designed for computer science students and professionals. This tool addresses such challenges by offering a beginner-friendly environment with guided assistance, enabling users to experiment and understand CNN architectures without requiring extensive programming knowledge. As users proceed to conduct more intense training on larger datasets, they can seamlessly transition to traditional coding workflows by exporting the generated Python code. This dual-purpose approach ensures that the application is both an educational resource and a practical development assistant.

1.5 Project

This project focuses on developing a web-based CNN model development platform designed to accommodate users of varying levels of expertise. For beginners, it will provide interactive tooltips, feedback on layer compatibility, parameter validation, and dataset metadata inspection, offering guidance in CNN model construction. These features will enable users to

experiment with different model architectures and hyperparameters in an intuitive and visual manner. For intermediate and experienced users, the platform will include automatic Python code generation upon model creation through the user interface, allowing for a seamless transition into traditional machine learning workflows. By reducing manual coding effort and minimizing potential errors, the tool will enhance both educational and practical aspects of CNN development, making it an accessible yet powerful resource for users across multiple domains.

Chapter 2: Technical Background

Summary

This chapter outlines the key technical concepts relevant to this project, providing an overview of the fundamental technologies and frameworks that will be used in its development.

2.1 Convolutional Neural Networks

A Convolutional Neural Network is a type of artificial neural network designed specifically for image classification tasks. Unlike traditional neural networks, CNNs are structured to recognize spatial hierarchies in images, allowing them to detect patterns such as edges, textures, and objects. The architecture of these networks consists of multiple layers, each serving a specific function in the training process. Convolutional layers apply filters to extract important features, while activation functions such as ReLU introduce non-linearity to capture complex patterns. Pooling layers reduce spatial dimensions, fully connected layers translate extracted features into class predictions, and the output layer determines the final classification result. CNNs are trained using large datasets of images and the choice of architecture, whether fully custom or based on well-known models like AlexNet or MobileNet, significantly impacts a CNN's performance. Additionally, adjusting key hyperparameters such as filter size, learning rate, and batch size influences how a model learns, generalises, and classifies images. By optimizing these factors, CNNs can be adapted to a wide range of real-world applications, from medical imaging to autonomous driving.

2.2 TensorFlow

TensorFlow is an open-source deep learning framework developed by Google for building and training machine learning models. It provides a computational backend optimized for handling large scale numerical operations, with support for both CPU and GPU acceleration. It integrates with Python seamlessly and includes high level APIs such as Keras, which simplify the implementation of neural networks like CNNs by providing predefined functions for layer construction, activation functions, and optimization algorithms. It handles the key processes of training such as backpropagation and weight optimization while also offering tools for data preprocessing and model evaluation. Due to its efficiency and scalability, TensorFlow is widely used in both research and industry applications. [1]

2.3 Django

Django is a high-level Python web framework that follows the Model-View-Template architectural pattern and provides a range of built-in features, such as an ORM, authentication, and security mechanisms. Its seamless integration with TensorFlow ensures that there is a cohesive development environment within Python. Additionally, its widespread use, extensive documentation, and scalability make Django a reliable foundation for implementing the application. [5]

2.4 React

React is a popular JavaScript library for building user interfaces, particularly well-suited for single page applications. It utilises a modular component-based architecture, enabling updates to the user interface without requiring full page reloads. React simplifies the implementation of dynamic components, making it easier to manage interactive elements and real time updates, this suiting the applications needs well. [6]

2.5 Technology Stack Integration

The combination of React, Django, and Tensorflow will allow for the development of this project. React enables the creation of a dynamic and responsive user interfaces. Django serves

as the backend framework, managing the flow of data and integrating the various components of the application. Tensorflow, incorporated into Django, powers the backend APIs, facilitating CNN model training. Together, these technologies form a cohesive system that supports the implementation of this project.

Chapter 3: Solution

Summary

This chapter outlines the solution, covering user stories, requirements, wireframes, and implementation.

3.1 User Stories

To ensure that the application meets the needs of the intended users, the following user stories outline the key functionalities.

3.1.1 Beginner Users

1. As a beginner, I want to upload a dataset and inspect its contents so that I can understand the data before training a model.
2. As a beginner, I want to view dataset metadata (e.g., number of classes, if it has the correct train/test split, type of colour channel) so that I can verify its suitability for CNN training.
3. As a beginner, I want to remove a dataset so that I can manage the number of datasets on the application.
4. As a beginner, I want to be able to select a dataset that I have uploaded to train a model.
5. As a beginner, I want to be able to add and remove layers using buttons in the model builder so that I can modify the architecture without needing to write code.
6. As a beginner, I want to edit hyperparameters (e.g., filter size, activation functions, learning rate) so that I can experiment with CNNs and build up an intuition for effective architectures.
7. As a beginner, I want invalid hyperparameter values to be flagged and layer compatibility to be validated upon saving so that I can avoid making mistakes.
8. As a beginner, I want tooltips explaining each CNN layer so that I can learn about their function without needing to refer to external documentation.
9. As a beginner, I want to load a sample CNN model so that I can see a working example before making my own modifications.
10. As a beginner, I want to be able to view validation results so that I can understand how my model is performing.
11. As a beginner, I want to be able to take note of validation results during a session without needing to use an external application.

3.1.2 Intermediate and Advanced Users

12. As an experienced user, I want a button that will generate the Python code so that I can quickly implement a TensorFlow model.
13. As an experienced user, I want the model building interface to be faster than manually typing the code so that I can prototype architectures efficiently.

3.2 System Requirements

3.2.1 Functional Requirements:

The application must allow users to upload, inspect, and manage datasets, providing useful metadata such as the number of classes, train/test split validation, and colour channel type. Users must be able to build a CNN model through an intuitive interface, adding or removing layers via buttons and modifying hyperparameters and receiving validation feedback upon save. The system should also provide tooltips for educational guidance, load sample CNN models into the model building interface for reference, and display validation results after training, as well as a notepad for recording results and other information. Advanced users must have the ability to generate and export Python code corresponding to the model created in the UI.

3.2.2 Non-Functional Requirements:

The application should offer a clean and intuitive user interface. It must provide responsive performance, enabling smooth model building, and validation without significant delays. Error messages and validation checks should be clear and informative. The system should maintain a consistent and professional aesthetic remaining in line with colourblind and accessibility guidelines, as well as ensuring usability across various screen sizes.

3.3 Wireframe

Below are wireframe images of the application, illustrating its layout and functionality at different stages of use.

3.3.1 Initial State (Before Model Creation)

The first wireframe represents the application's interface before the user begins building a model. On the left side, the dataset panel displays all uploaded datasets, allowing users to inspect, select, upload, or remove datasets. In the centre, the model builder interface provides options to either start building a CNN model layer by layer or load a pre-existing architecture. At this stage, the interface remains minimal, awaiting user interaction. At the bottom, there is a training response display, which will later be populated with results, and a notepad section for users to take notes during the process.

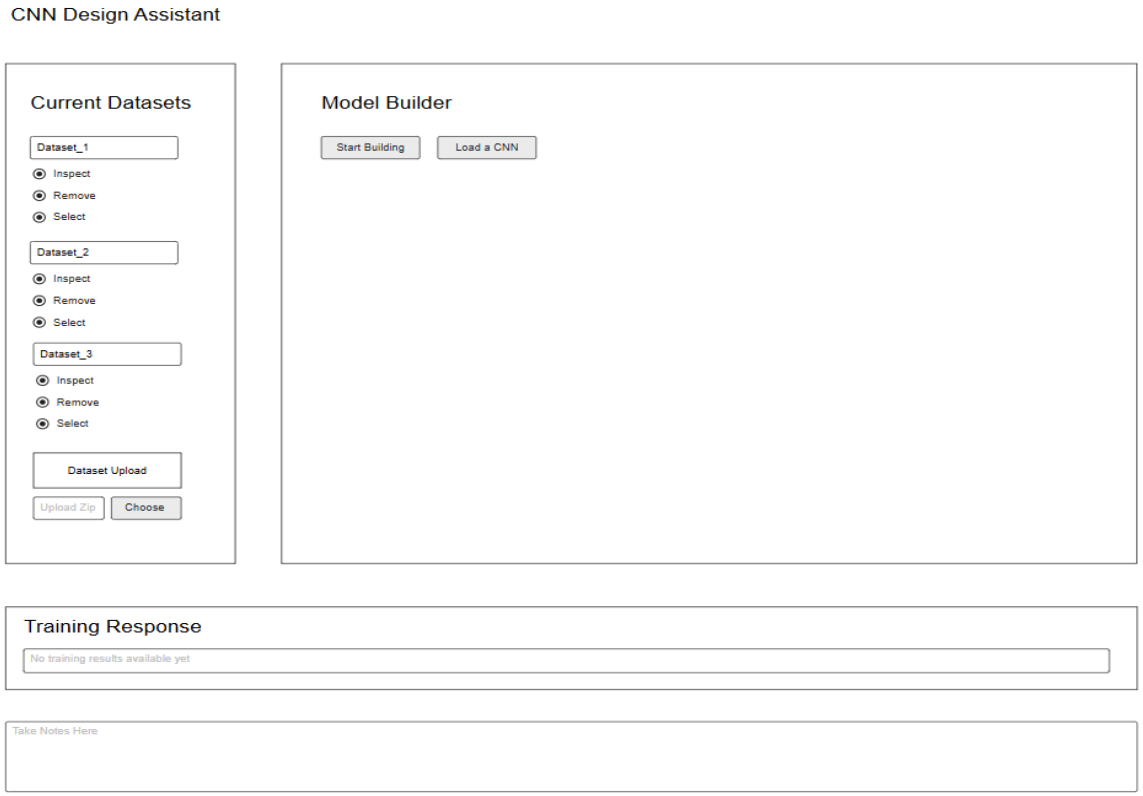


Figure 3-1 Wireframe 1

3.3.2 Model Building and Training Interface

The second wireframe shows the application after the user has begun creating a CNN model. Users can add a new layer through pressing the plus button and then selecting the layer type from a dropdown list. Layer-specific parameters such as filter size, kernel size, activation function, and pooling size can then be modified as needed. Layers can be removed or added anywhere within the model, except for the input layer, which remains fixed as it is a required component in all CNN architectures. Global hyperparameters, such as optimizer, loss function, and learning rate, can also be configured. Once the model is set up, users can save the model, train it using the selected dataset, or generate Python code for further use in a traditional development environment. Training results will appear in the training response display, while error messages related to layer compatibility or hyperparameter settings will be shown within the model builder. The notepad section remains available for users to document observations and insights throughout the process.

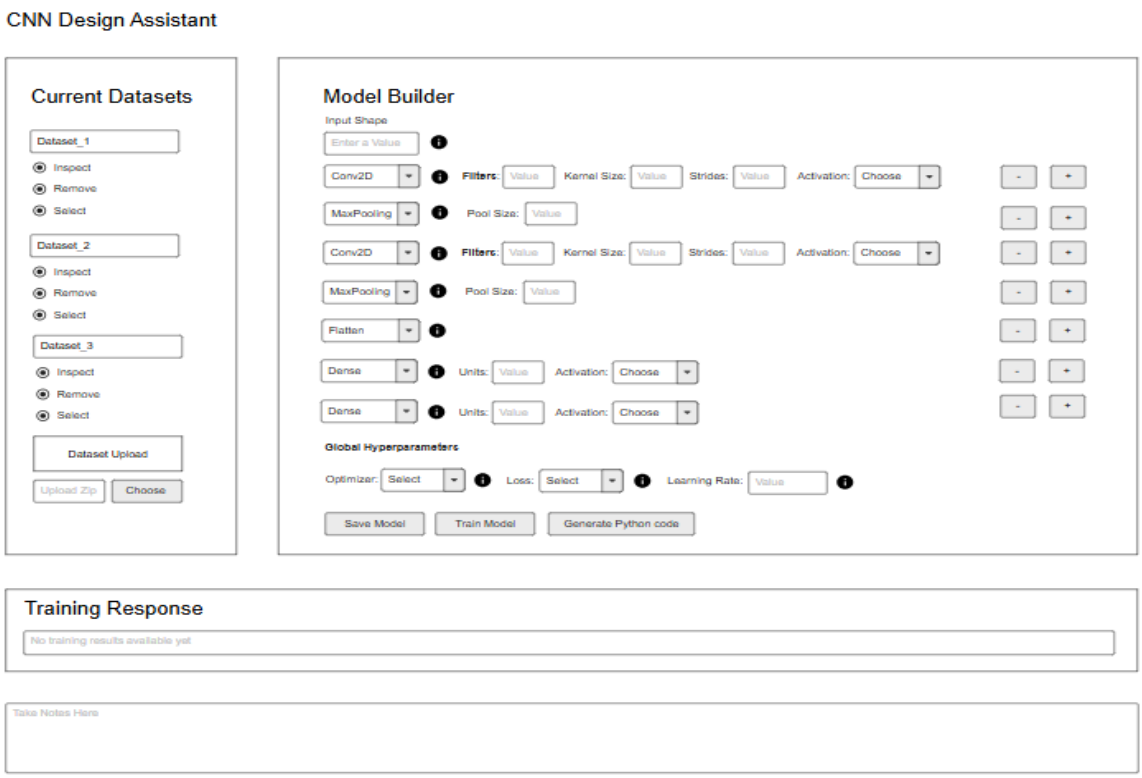


Figure 3-2 Wireframe 2

3.4 High Level System Architecture

The diagram below provides an overview of the applications structure, illustrating the interaction between the frontend, backend, dataset storage and APIs.

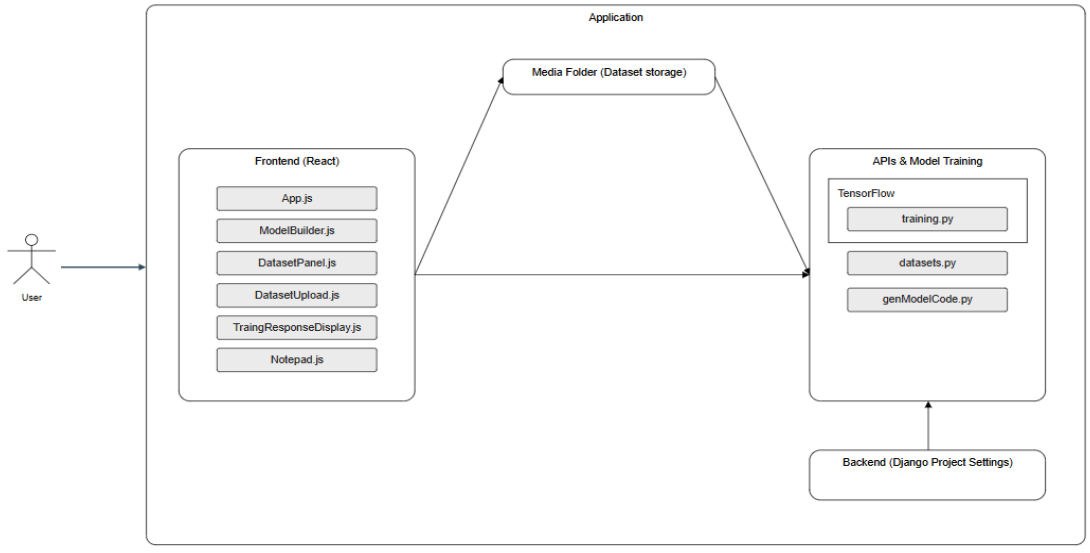


Figure 3-3 System Architecture Diagram 1

The frontend provides the UI for dataset management, model building, and training execution, communicating with the backend via API requests. Key components include App.js, which manages the state of ModelBuilder.js, DatasetPanel.js, DatasetUpload.js, and TrainingResponseDisplay.js.

The backend primarily handles project settings, including API routing and database configuration. Core functionalities, such as dataset management, model training, and code generation, are handled within the cnn_assistant module, referred to as APIs & Model Training in Figure 3-3. This includes datasets.py for handling dataset operations (uploading, removing, inspecting, and fetching), training.py for TensorFlow-based model training, and genModelCode.py for Python code generation. Dataset storage is managed within the media folder, ensuring accessibility to the required APIs.

Workflow Summary

1. The user uploads a dataset via dataset panel on the frontend
2. The dataset is processed by the dataset API and stored in the application's media folder.
3. The user creates a model in ModelBuilder.js.
4. The model, represented in JSON format, is sent to the train.py endpoint in cnn_assistant, along with the selected dataset's ID.
5. In train.py, the JSON object is parsed, the dataset is loaded from the media folder, and the model is built layer by layer. Training and validation are then executed
6. Upon successful training, the results are returned to TrainingResponseDisplay.js on the frontend.

If the user opts to generate Python code for their model, the JSON object representing the model (excluding the dataset ID) is sent to the genModelCode.py endpoint. There, the object is parsed, reconstructed into a model, and returned and displayed just below the model builder in plaintext format.

3.5 Frontend Implementation

3.5.1 Application (App.js)

App.js serves as the core component of the frontend, acting as the main entry point for the application. It maintains the global state using the useState hook and integrates the DatasetPanel, DatasetUpload, ModelBuilder, TrainingResponseDisplay, and Notepad components. Key state variables managed in this file include datasets, selectedDataset, modelConfig, trainingStatus, and trainingResponse. Upon initialization, the useEffect hook triggers the fetchDatasets function, which retrieves available datasets from the backend via an API request. User interactions are handled through the following functions:

- handleSelectDataset(dataset) – Updates the state with the currently selected dataset.
- saveModelConfig(config) – Stores the user's model configuration settings.
- trainModel(modelConfig) – Initiates the model training process by sending a POST request to the backend with the selected dataset ID and model configuration. It includes error handling to ensure a dataset is selected before training.
- getModelCode(config) – Requests the backend to generate and return the corresponding model code based on the provided configuration.

- clearTrainingResponse() – Resets the training response data to allow for a fresh training session.

3.5.2 Viewing Datasets (DatasetPanel.js)

The DatasetPanel.js component is responsible for displaying and managing datasets within the application. It provides users with options to inspect, select, and remove datasets while fetching relevant metadata and file structures from the backend. Upon user interaction, the following functions handle dataset operations:

- handleInspect(dataset) – Fetches and displays the file structure and metadata of a selected dataset by making a GET request to the backend.
- handleClose() – Clears the displayed file structure and metadata, closing the dataset inspection view.
- handleRemove(datasetId) – Sends a DELETE request to the backend to remove a dataset from the system. A confirmation prompt ensures users do not delete datasets unintentionally. The function also prevents multiple requests by disabling the remove button during the deletion process.
- handleSelectDataset(dataset) – Toggles dataset selection, allowing users to either select or deselect a dataset for further operations.

The component renders a list of available datasets, each with Inspect, Remove, and Select Dataset buttons.

3.5.3 Uploading Datasets (DatasetUpload.js)

The DatasetUpload.js component allows users to upload datasets in ZIP format. It manages file selection, initiates the upload process via an API request, and provides feedback on the upload status. The key functionalities are:

- handleFileChange(e) – Updates the state with the selected file.
- handleUpload(e) – Sends the dataset to the backend using a POST request. It prevents uploads without file selection and displays success or error messages accordingly.

The UI consists of a file input field and an upload button, which is disabled during an ongoing upload. Upon successful upload, the dataset list is refreshed.

3.5.4 Model Building Interface (ModelBuilder.js)

The ModelBuilder.js component provides the interactive interface for users to construct and configure models, allowing for layer customization, model validation, and hyperparameter tuning.

(i) Model Initialization & Construction

The component allows users to start model construction in two ways:

- Manually: The initialiseModel() function is triggered when the Build Model button is clicked. It sets modeInitialised to true and automatically adds a Conv2D layer as the first layer.
- Using a predefined structure: The loadGenericCNN() function creates a default CNN model with two Conv2D layers, two MaxPooling2D layers, a Flatten layer, and two Dense layers.

Adding layers dynamically is done through the addLayer(index), removeLayer(index), and updateLayer(index, field, value) functions. The addLayer function inserts a new layer at a specified index. When the Add button is pressed on an existing layer, a new layer is added directly below it in the model architecture, while removeLayer deletes an existing layer. The updateLayer function modifies properties of a selected layer.

(ii) JSON Representation of the Model

When a model is configured, it is stored as a JSON object, which contains:

- Input shape: A string representing the model's input dimensions.
- Layers: An array where each object represents a layer, containing properties such as type, filters, units, kernel size, activation, etc.
- Global hyperparameters: Including optimizer, loss function, learning rate, and epochs.

This structured JSON format ensures that the model can be easily passed to the backend for processing and training.

(iii) Layer & Model Validation

Before saving or training a model, the validateLayerCompatability() function ensures that the model follows structural constraints and that the layers are in the correct order. These include:

- The first layer must be a Conv2D to properly process image data.
- A Conv2D layer cannot follow a Dense layer, as convolutional layers require multi-dimensional input.
- A Dense layer must be preceded by a Flatten layer to convert multi-dimensional data into a one-dimensional vector.
- A MaxPooling2D layer must follow either a Conv2D or BatchNormalization layer to ensure valid down sampling.
- A Dropout layer cannot be the first layer, as regularization should only be applied after meaningful feature extraction.
- A BatchNormalization layer must follow a Conv2D or MaxPooling2D layer to normalize activations correctly.
- The **f**inal layer must be a Dense layer, ensuring the correct output shape for classification tasks.

Beyond layer ordering, numerical input validation ensures that kernel sizes, strides, and pooling sizes follow the correct format, such as '3x3' or '2x2'. Additionally, learning rate and epoch values are checked, displaying a warning if the number of epochs exceeds 15, reminding the user that a higher number of epochs will require access to a more powerful machine if used locally.

(iv) Tooltips for Layers

Tooltips appear when a user hovers over an input field or dropdown selection. For example, hovering over a Conv2D layer displays a tooltip explaining that it extracts spatial features from images through kernel operations. Similarly, a Dense layer tooltip informs users that it is a fully connected layer used for classification tasks. These are present for all the layers available to the user

(v) Training & Model Code Generation

Once the model is validated, users can train it by clicking Train Model. The model configuration, along with selected hyperparameters, is sent as a JSON payload to the backend via an API request. For users who want to generate Python code for the model, the handleGetModelCode() function requests a Python script representation of the current architecture from the backend. The generated code is displayed inside a modal window, where users can either copy it to the clipboard or close the modal. A progress indicator prevents multiple concurrent training requests, ensuring system efficiency and avoiding unnecessary computational load.

(vi) User Feedback & Validation Messages

Validation messages are displayed in a section below the model configuration area. If there are missing required layers or incorrect ordering, error messages appear in red to inform users of the issue. If the model is successfully trained, an alert notifies the user, and the training response data is displayed within the TrainingResponseDisplay.js component. Additionally, if users attempt to generate Python code but encounter an issue, an error message is shown, and they are prompted to correct their model before trying again.

For the full implementation of all frontend components mentioned in this section, see Appendix 1

3.6 Backend Implementation

3.6.1 Handling Datasets (datasets.py)

datasets.py contains the backend functionality for managing uploads, retrieval, structure analysis, and deletion. It provides API endpoints that interact with the media folder to store dataset metadata and handle filesystem operations such as extracting, organizing, and validating datasets. The endpoints include the following:

(i) Dataset Upload

DatasetUploadView handles dataset retrieval and uploading within the backend. It allows users to upload datasets in ZIP format, which are then extracted and stored in the media folder. Functions include:

- get(request) – Retrieves all datasets stored in the database and returns their ID, name, and root directory.
- post(request) – Processes dataset uploads by:
    - Saving the uploaded ZIP file temporarily.
    - Extracting its contents into a dedicated folder inside the datasets/ directory.
    - Storing the dataset details in the database for future access.
    - Removing the temporary ZIP file after extraction.

If no file is provided, an error response is returned.

(ii) Dataset Structure

The DatasetStructureView allows users to inspect the internal structure of a dataset and retrieve metadata. Upon a user pressing the inspect button, the following operations occur:

- get(request, dataset_id) – Scans the dataset's directory and returns:
    - Folder structure, showing how files are organized.
    - Image metadata, including the number of images, colour channels, and train/ test class distributions.
    - Validation checks, identifying train/test mismatches and inconsistent image sizes within classes.

(iii) Dataset Structure

The DatasetDeleteView allows users to remove datasets from the filesystem. The following functions handle dataset deletion:

- delete(request, dataset_id) – Deletes the dataset's directory and its corresponding database entry.
    - Ensures that the media root directory itself cannot be deleted.
    - Removes empty parent directories only if they are no longer needed.
    - Handles permission errors gracefully to prevent unintended failures.

The system includes confirmation prompts and safety checks to avoid accidental mass deletions.

3.6.2 Handling Model Training (training.py)

training.py handles the loading of datasets from the filesystem, dynamic model creation, and model training using TensorFlow. It includes functions for processing datasets, constructing models based on user input, and executing the training process. The key functions and endpoints are as follows:

(i) Dataset Loading

The load_dataset function is responsible for retrieving dataset information from the database and preparing it for model training. Upon execution, it performs the following operations:

- Retrieves the dataset using the provided dataset ID and locates the root directory.
- Verifies and corrects the dataset folder path, ensuring the correct train/test directory structure is used.
- Determines the colour mode based on the expected number of channels (RGB or grayscale).
- Uses TensorFlow's image_dataset_from_directory() to load training and testing datasets, automatically inferring labels and formatting the data into batches.

If the dataset does not exist, an error is raised, and any unexpected issues are logged for debugging.

(ii) Building the model dynamically

The build_dynamic_model function constructs the neural network model dynamically based on the user-provided configuration within the JSON object. Upon execution, it:

- Initializes a Sequential Keras model and adds an Input Layer with the specified shape.
- Iterates through the list of layer configurations, adding layers dynamically based on their type.
- Configures layer-specific parameters, such as filters, kernel_size, activation, and dropout rate.
- Ensures the final layer is a Dense layer with units matching the number of dataset classes.
- Compiles the model using the specified optimizer and loss function, applying the user-defined learning rate.

If the configuration is incorrect (e.g., missing a required layer or invalid activation function), an error is returned.

(iii) Training the model

The train_model function is an API endpoint that handles the entire training workflow. Upon receiving a POST request, it:

- Extracts dataset ID, input shape, hyperparameters, and layer configurations from the request data.
- Loads the dataset using load_dataset(), ensuring it is correctly formatted for training.
- Determines the number of classes by analysing the dataset structure.
- Validates that the final layer of the model matches the number of dataset classes, preventing classification errors.
- Builds the model dynamically using build_dynamic_model().
- Initiates model training using the dataset, running for the specified number of epochs.
- Evaluates the trained model on the test dataset, returning validation accuracy and loss in the response.

If any errors occur during training, they are logged and returned to the user. [Appendix 6]

3.6.3 Generating Python code (genModelCode.py)

gen_model_code.py is responsible for generating the model code from the JSON configuration provided by the frontend. It provides an API endpoint that parses the model structure and hyperparameters and converts them into a plain text Python function that defines and compiles the specified neural network.

(i) Generating Model Code

The generate_model_code function constructs a Python script based on the user-defined model configuration. Upon execution, it:

- Initializes a Sequential Keras model and adds an Input Layer with the specified shape.

- Iterates through the list of layer configurations, adding layers dynamically based on their type and parameters.
- Ensures that the final layer structure aligns with classification requirements.
- Compiles the model using the specified optimizer and loss function, applying the user-defined learning rate.
- Formats the generated Python code as a structured function, making it ready for execution.

The output is a plain text function definition, allowing users to copy and use the generated model script directly in Python.

For the full implementation of all backend components mentioned in this section, see Appendix 1

Chapter 5: Evaluation

Summary

This chapter evaluates the system's correctness, functionality, and performance.

5.1 Solution Verification

Three datasets were chosen for training: a sports ball dataset from Kaggle, a nuts and bolts dataset, and the MNIST dataset. Each was uploaded, processed, and trained within the application. To validate its effectiveness, the same datasets were also trained separately using a traditional machine learning workflow in Spyder with TensorFlow. The results were consistent across both methods, confirming that the system correctly follows the standard deep learning workflow of dataset preparation, model building, training, and evaluation. This demonstrated that the application is fully capable of training user-defined models. Additionally, the generated Python model code was tested independently, further proving its correctness and ensuring that models constructed within the application could function as expected when used externally.

5.1.1 Workflow Example

The following images demonstrate the functionality of the core features of the system.

In the first screenshot, we can see the process of uploading a dataset within the application. The interface allows users to select a dataset. Once uploaded, it appears in the dataset panel, ready for inspection and model training.

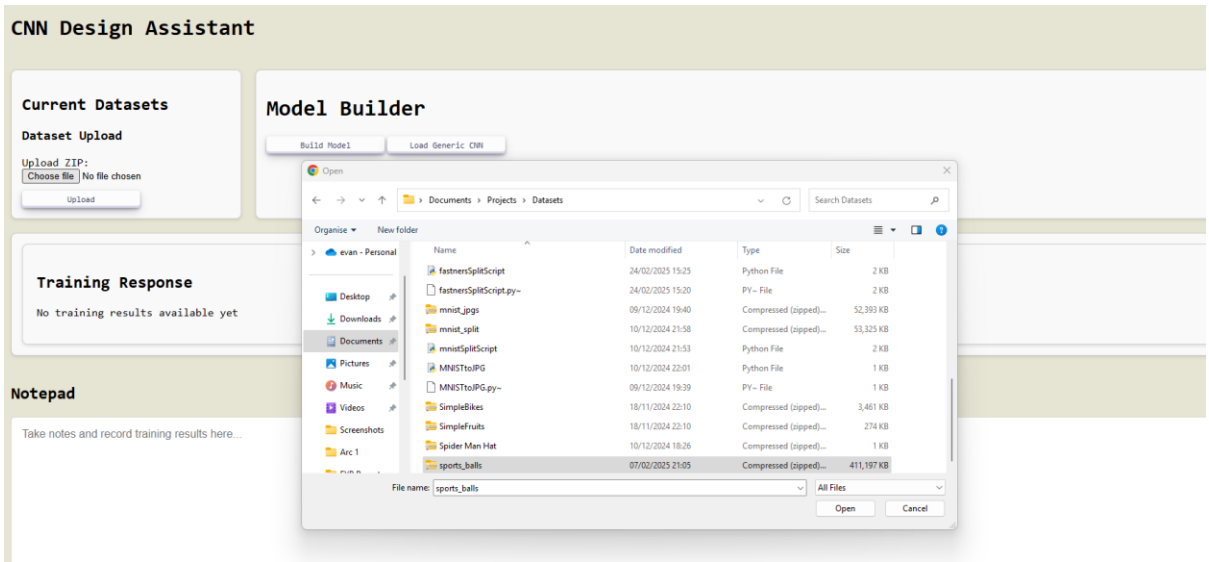


Figure 5-1 Image of uploading a dataset

On the left, we see how multiple datasets are displayed in the application. On the right, we see how the dataset inspection panel provides an overview of a chosen dataset, showing its metadata and file structure.

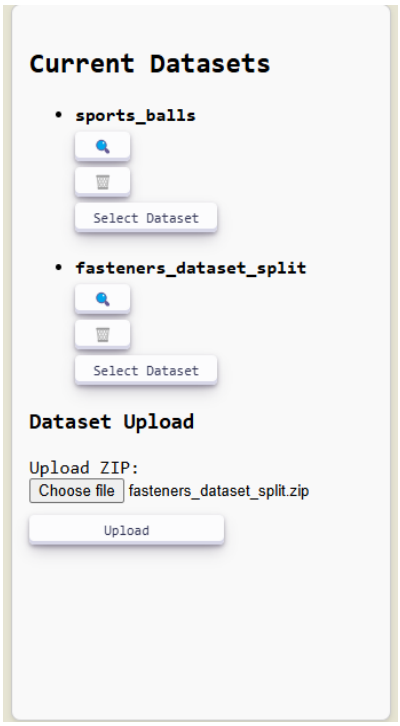


Figure 5-2 Dataset Panel 1

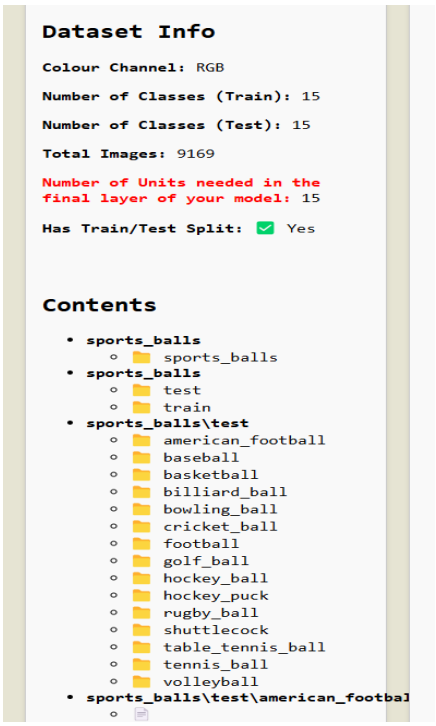


Figure 5-3 Dataset Panel 2

Here, we can see a model that has been successfully created and saved within the application

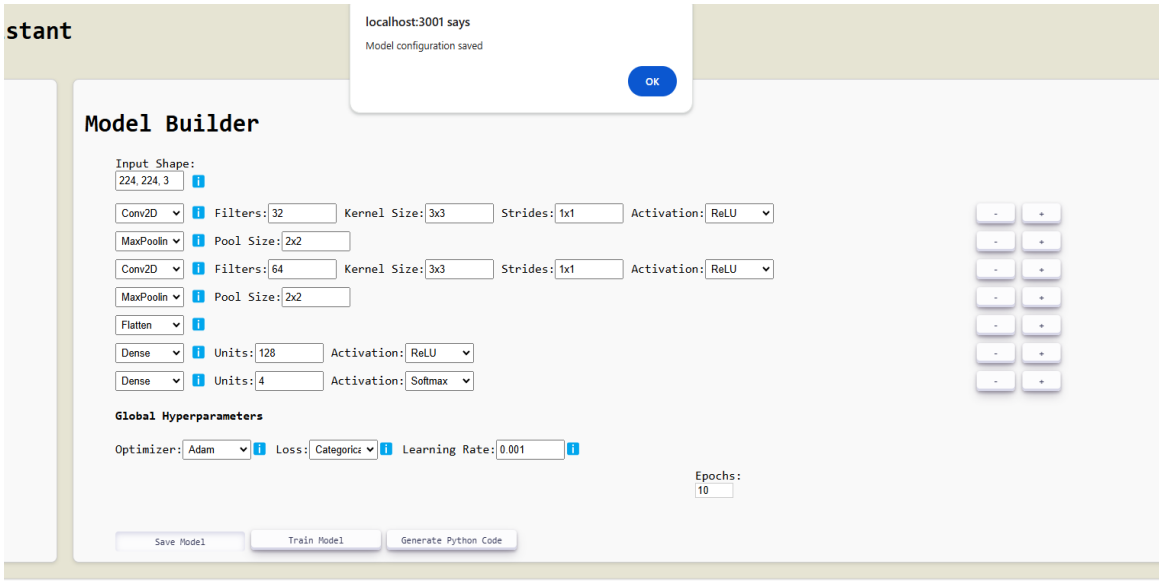


Figure 5-4 Model Builder

Below, we see the training results of the fastener dataset within the application, displaying the loss and accuracy achieved using the model shown above.

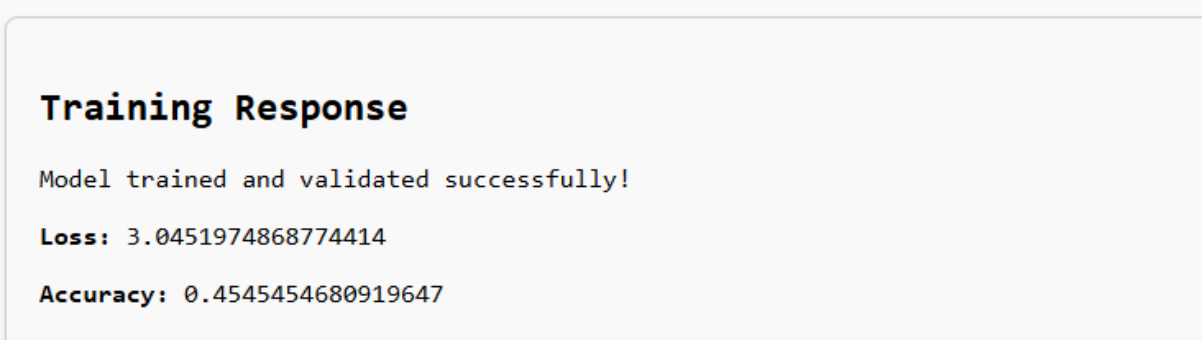
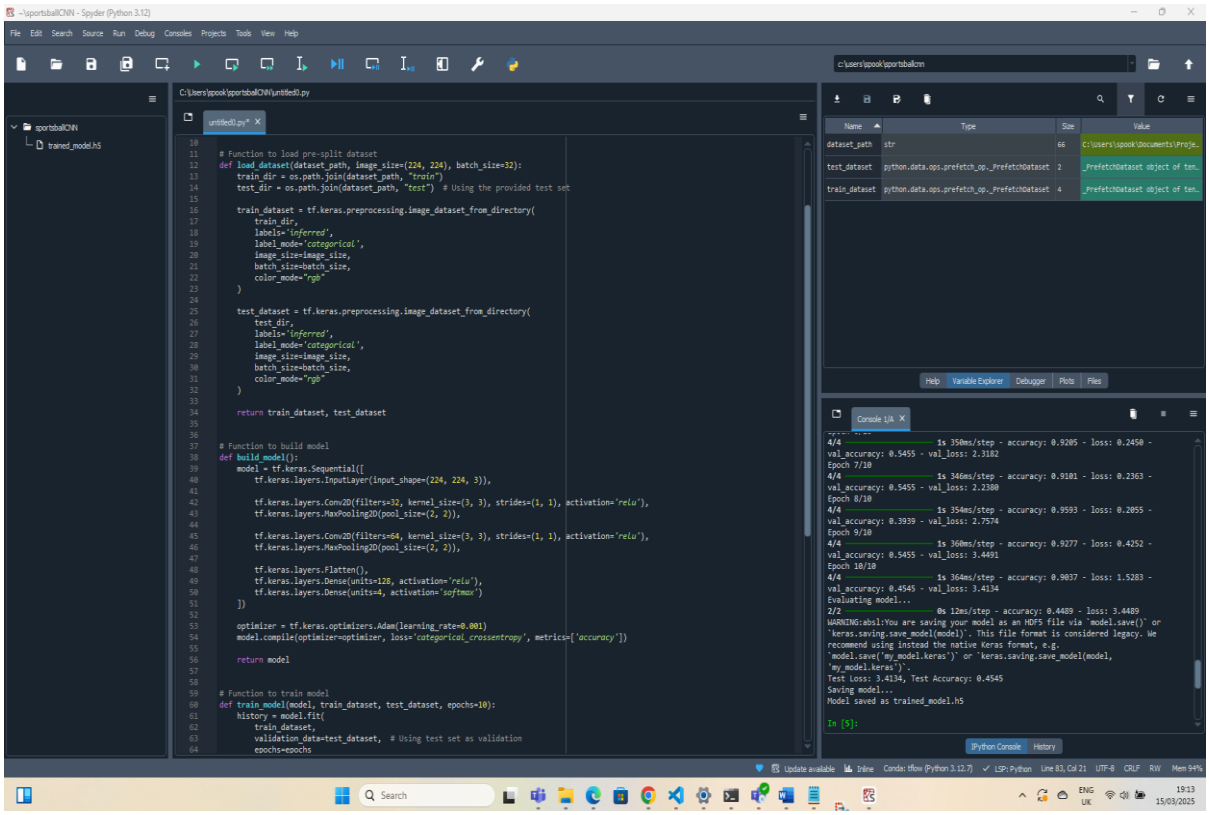


Figure 5-5 Training response

To validate the application's effectiveness, the generated model code will be used in a traditional workflow to produce comparable results.

Figure 5-6 Model code generated

Here, we see the model code copied and pasted into Spyder, along with the necessary functions for loading the dataset.



```
# Function to load pre-split dataset
def load_dataset(dataset_path, image_size=(224, 224), batch_size=32):
    train_dir = os.path.join(dataset_path, "train")
    test_dir = os.path.join(dataset_path, "test")  # Using the provided test set

    train_dataset = tf.keras.preprocessing.image_dataset_from_directory(
        train_dir,
        labels='inferred',
        label_mode='categorical',
        image_size=image_size,
        batch_size=batch_size,
        color_mode="rgb"
    )

    test_dataset = tf.keras.preprocessing.image_dataset_from_directory(
        test_dir,
        labels='inferred',
        label_mode='categorical',
        image_size=image_size,
        batch_size=batch_size,
        color_mode="rgb"
    )

    return train_dataset, test_dataset


# Function to build model
def build_model():
    model = tf.keras.Sequential([
        tf.keras.layers.InputLayer(input_shape=(224, 224, 3)),

        tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), activation='relu'),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),

        tf.keras.layers.Conv2D(filters=64, kernel_size=(3, 3), strides=(1, 1), activation='relu'),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),

        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(units=128, activation='relu'),
        tf.keras.layers.Dense(units=4, activation='softmax')
    ])

    optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

    return model


# Function to train model
def train_model(model, train_dataset, test_dataset, epochs=10):
    history = model.fit(
        train_dataset,
        validation_data=test_dataset,  # Using test set as validation
        epochs=epochs
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 350ms/step - accuracy: 0.9205 - loss: 0.2450 -
val_accuracy: 0.5455 - val_loss: 2.3182
Epoch 7/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 346ms/step - accuracy: 0.9101 - loss: 0.2363 -
val_accuracy: 0.5455 - val_loss: 2.2380
Epoch 8/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 354ms/step - accuracy: 0.9593 - loss: 0.2055 -
val_accuracy: 0.3939 - val_loss: 2.7574
Epoch 9/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 360ms/step - accuracy: 0.9277 - loss: 0.4252 -
val_accuracy: 0.5455 - val_loss: 3.4491
Epoch 10/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 364ms/step - accuracy: 0.9037 - loss: 1.5283 -
val_accuracy: 0.4545 - val_loss: 3.4134
Evaluating model...
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.4489 - loss: 3.4489
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
`keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')` or `keras.saving.save_model(model,
'my_model.keras')`.
Test Loss: 3.4134, Test Accuracy: 0.4545
Saving model...
Model saved as trained_model.h5

In [5]:
```

Figure 5-7 Traditional workflow 1

The main execution, including the path to a copy of the same dataset, ensuring consistency with the dataset used in the application.

```
# Main execution
if __name__ == "__main__":
    dataset_path = r"C:\Users\spook\Documents\Projects\Datasets\fasteners_dataset_split"  # Set this to the actual dataset path

    train_dataset, test_dataset = load_dataset(dataset_path)
    model = build_model()

    print("Training model...")
    train_model(model, train_dataset, test_dataset, epochs=10)

    print("Evaluating model...")
    evaluate_model(model, test_dataset)

    print("Saving model...")
    save_model(model)
```

Figure 5-8 Traditional workflow 2

Here we see the training results with the same number of epochs, ensuring a direct comparison with the results from the application.

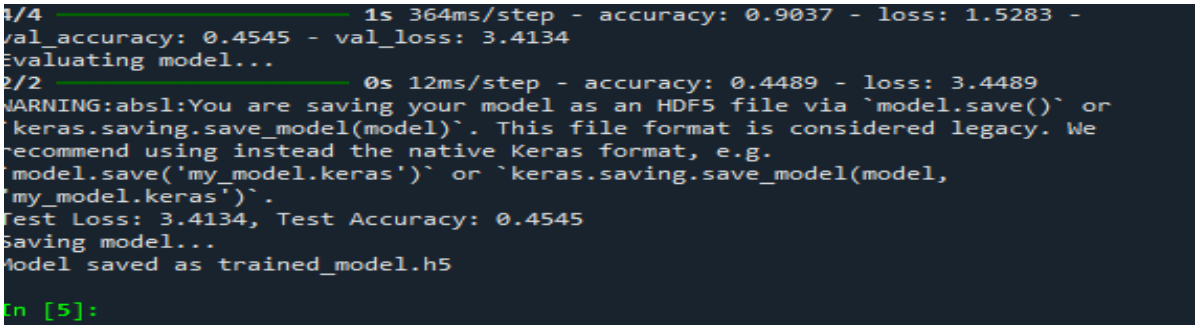


Figure 5-9 Training results

Using the given dataset in the traditional workflow, we obtain a test accuracy of 0.4545. The application produces the same result, albeit with extended decimal precision. This confirms that the application functions correctly and aligns with standard deep learning workflows.

5.2 Software Design and Verification

The system's design and functionality were validated through a development-driven testing approach, where testing was continuously integrated into the implementation process. API endpoints were tested directly to confirm that dataset uploads, metadata extraction, and training execution functioned correctly. Responses were manually verified to ensure proper data flow between the frontend, backend, and database. A key advantage of this approach was its ability to rapidly identify and fix errors, allowing for quick iterations and adjustments without the overhead of formal test cases. Issues were immediately logged, debugged, and corrected, ensuring that features were functional before moving on to the next stage of development. Comprehensive exception handling and try-catch blocks were implemented across the system to improve robustness. These measures ensured that unexpected failures, such as missing datasets, invalid user inputs, or failed API requests, were caught and handled gracefully without crashing the system. Users also contributed to validation by interacting with the system, confirming that it behaved correctly in real-world scenarios. While this approach is effective for development, structured testing would further enhance reliability, particularly for edge cases that may not have been encountered during hands-on testing.

5.3 Functional Testing through Execution

Many issues were uncovered and resolved through real-time testing of the application. One example of an improvement was ensuring that users could not start multiple training processes simultaneously, preventing excessive resource usage. Restrictions were also implemented to allow only one dataset upload or deletion at a time, avoiding potential conflicts. Initially, dataset deletion was handled recursively, which unintentionally removed entire directories. This was corrected by modifying the system to delete only the dataset's root folder while preserving higher-level directory structures. These and many other refinements, identified during continuous testing, contributed to the system's final stable and reliable state**.**

5.4 User Testing and Validation

The system was tested by real users who uploaded datasets, built models, and executed training. Their feedback confirmed that the interface was intuitive and that they were able to complete the machine learning workflow without issues. The system produced expected results, with no critical errors preventing dataset uploads, training, or result analysis, demonstrating that the application is both functional and user-friendly. Datasets were also processed and trained using a traditional TensorFlow workflow utilising the model code generated by the system, yielding consistent results that confirmed the code generated works as expected. Additionally, the system was assessed using Google Chrome's automatic accessibility test, which verified that it meets essential accessibility standards, ensuring a navigable and inclusive user experience. Through both real-world usage and direct validation against established workflows, the system was shown to be reliable, producing correct results without major failures. The system is functionally sound and performs well for its intended purpose. However, the application would benefit from additional structured testing and a more focused effort on identifying bugs through exploratory user testing, as there are likely still unknown issues that have yet to be discovered.

5.6 Critical Analysis

This section evaluates specific components of the system, highlighting strengths, weaknesses, and areas for improvement.

5.6.1 Dataset Upload

The dataset upload functionality works reliably, allowing users to add datasets without major issues. However, a potential improvement would be implementing validation checks before moving the dataset to the media folder. Currently, there is no verification of whether the uploaded file is a valid dataset before it is processed. Adding a preliminary validation step would prevent corrupted or incompatible files from being stored, improving efficiency and preventing unnecessary disk usage.

5.6.2 Dataset Structure View

The dataset structure view is one of the weaker areas of the application. While it correctly walks through the dataset directories and lists their contents, users cannot click on individual images to view them. This limitation arises from the current approach, which only retrieves file paths rather than rendering image previews. Additionally, due to the recursive nature of walking through datasets, inspecting and displaying the full contents of large datasets can take up to 30 seconds, making it inefficient for particularly large datasets. To improve performance, metadata retrieval could be cached to reduce processing time when inspecting datasets repeatedly. Alternatively, metadata could be processed during dataset upload and stored separately, allowing for much faster access instead of reprocessing the dataset structure on demand.

5.6.3 Training Process (training.py)

The training process functions as intended, allowing users to build and train models directly within the application. However, due to the complexity of CNN development, some aspects had to be hardcoded. For example, batch size is currently predefined rather than user-configurable, as the project does not handle dataset preprocessing in depth. Additionally, BatchNormalization is implemented as a separate layer rather than being integrated within a Conv2D layer, which is typically the recommended approach for optimal performance. These compromises and many more were made to maintain simplicity and usability, ensuring that users can construct and train models without excessive configuration overhead. While the system does its job well, advanced users may require greater flexibility in model and dataset processing.

5.6.4 Code Generator

The code generator effectively produces Python model code based on user-defined configurations. It successfully converts models built in the UI into executable TensorFlow code, making it easy for users to export and use their models outside of the application. However, the generator is entirely dependent on the ModelBuilder component, meaning any structural changes or limitations in ModelBuilder directly impact the code output.

5.6.5 Model Building

The model-building interface provides an intuitive way for users to design CNN architectures without writing code. It allows for quick experimentation and model creation faster than manual coding. However, some improvements could enhance the user experience. One potential addition is warnings for overly large models, such as excessive layers or redundant repeating layers, helping users avoid inefficient architectures. Another major improvement

would be the ability to drag and drop layers, making model design more flexible and user-friendly. While the current approach works well, these enhancements would further streamline the process and make model construction even more accessible.

5.7 Considerations for Deployment

If the system were to be deployed in a production environment, additional improvements would be necessary to ensure scalability, security and reliability

5.7.1 Dataset Validation

As mentioned above, the system does not validate datasets beyond their basic structure. File integrity checks would need to be implemented to detect corrupted files before processing. Dataset size limits would also need to be enforced to prevent excessive storage consumption and ensure that training runs efficiently on available resources.

5.7.2 Training Process Management

At present, users cannot stop training once it has started. If a user unintentionally sets the number of epochs too high, the system will continue training until completion, consuming unnecessary resources. A feature should be implemented that allows users to stop training mid-process from the backend. Backend monitoring should also be introduced to detect and handle stalled or failed training sessions.

5.7.3 Security Considerations

For deployment, the system would need to adhere to OWASP security principles to prevent vulnerabilities such as unauthorized API access or malicious dataset uploads. Authentication and access control mechanisms should be implemented to restrict actions like uploading, training, and deleting datasets to authorized users only. Additionally, user inputs should be properly validated and sanitized to prevent injection attacks or unintended file manipulations.

Chapter 6: Conclusion

Summary

This chapter concludes the project, summarising its evaluation and findings.

6.1 Contribution to the State-of-the-Art

This project does not contribute to the state of the art in deep learning but improves accessibility and usability in CNN model development. It simplifies the workflow by integrating dataset management, model building, training, and code generation into an intuitive system. While it does not introduce new deep learning methodologies, it lowers the barrier to entry, making CNN development more structured and user-friendly without compromising functionality.

6.2 Results Discussion

The results confirm that the system effectively processes datasets, generates models, and executes training workflows, aligning with traditional deep learning methods. However, the generalizability of these results depends on the datasets used. The system was tested with structured image datasets but may require adjustments to handle more complex, unstructured data formats. Additionally, performance limitations arise with large datasets, as inspecting and structuring data can be slow. The development-driven testing approach ensured that the system is fully functional, as demonstrated by its ability to consistently perform the intended tasks without critical failures.

6.3 Project Approach

6.3.1 Research

The project began with a theoretical study of CNNs, followed by hands-on experience training models using traditional workflows. This process highlighted common pain points for beginners, which directly informed the user stories and system requirements. The goal was to create a tool that addressed these challenges by making the CNN development process more structured and intuitive.

6.3.2 Development and Use of Generative AI

Prior experience developing single-page applications in React helped streamline frontend development. Generative AI, particularly ChatGPT, became an integral part of the workflow for quickly upskilling in TensorFlow and Django. The typical workflow involved presenting ChatGPT with the required context for a particular problem, proposing an implementation strategy, and then requesting critiques and alternative approaches. This allowed for maximized learning while optimizing for the best solution. However, AI-generated code was often error-

prone and required extensive debugging and iteration. ChatGPT struggled with contextualizing multiple files, making it most useful for answering specific technical questions rather than managing complex multi-file interactions.

6.4 Future Work

Dataset preprocessing could be integrated directly into the dataset interface, allowing users to clean and format their data before training. Another improvement would be live training data visualization, displaying accuracy and loss per epoch on a graph, along with validation performance per class. This would help users identify potential outliers that are not being recognized effectively. The dataset panel could also be redesigned to support drag-and-drop image classification, allowing users to manually classify images and view model predictions. One notable omission from the project is image classification using a trained model, though this may be redundant since effective training is typically done in a traditional workflow, where users can leverage the generated model code instead. Additionally, as outlined in the previous chapter, security and deployment considerations, such as data validation and process management, would need to be addressed before deploying the system in a production environment.

6.5 Achievements

The system successfully delivers on its core objective: providing an interactive and functional CNN model-building experience. It allows users to upload datasets**,** build models, train them, and generate Python code, all within a web-based interface. The application effectively translates a traditional deep learning workflow into an intuitive system, making CNN development accessible and user-friendly.

References

TensorFlow Team, Google, *TensorFlow: An Open Source Machine Learning Framework*, [https://www.tensorflow.org/, accessed on: 13 March 2025].

TensorFlow Team, Google, *TensorFlow Tutorials*, [https://www.tensorflow.org/tutorials, accessed on: 13 March 2025].

Karpathy, A., Stanford University, *ConvNetJS: Deep Learning in Your Browser*, [https://cs.stanford.edu/people/karpathy/convnetjs/, accessed on: 13 March 2025].

TensorFlow Team, Google, *TensorBoard: Visualization Toolkit for TensorFlow*, [https://www.tensorflow.org/tensorboard, accessed on: 13 March 2025].

Django Software Foundation, *Django: The Web Framework for Perfectionists with Deadlines*, [https://www.djangoproject.com/, accessed on: 13 March 2025].

Meta, *React: A JavaScript Library for Building User Interfaces*, [https://react.dev/, accessed on: 13 March 2025].

Appendices

Appendix 1: Source Code Repository

Web-Based CNN Design Assistant

Frontend Components

- **Main Application:** App.js
- **Dataset Management:** DatasetPanel.js
- **Dataset Uploading:** DatasetUpload.js
- **Model Builder:** ModelBuilder.js
- **Training Response Display:** TrainingResponseDisplay.js
- **Notepad Component:** Notepad.js

Backend Components

- **Dataset Handling:** datasets.py
- **Model Training:** training.py
- **Code Generation:** genModelCode.py