# Deep Learning for Book Genre Classification using Amazon Book Covers

Niccolò Cibei, Tommaso Premoli

June 12, 2025

**Abstract**

This project investigates the use of deep neural networks to classify book genres based on their cover images. Using a filtered subset of the Amazon Book Covers dataset, we explore a convolutional architecture and evaluate its performance and scalability on progressively larger datasets. The entire pipeline is implemented in Python using TensorFlow and is available via Google Colab and GitHub.

## Contents

## 1 Introduction

In the context of large-scale data analysis, the ability to extract structured insights from unstructured sources such as images is increasingly relevant. This project addresses the task of classifying book genres based solely on cover images, using a deep learning approach. The dataset originates from the Amazon Books category, which provides both image URLs and metadata such as titles, categories, and descriptions.

The main objective is to build a Convolutional Neural Network (CNN) capable of accurately predicting the genre of a book from its cover, using scalable techniques that could, in principle, be applied to much larger datasets.

Key contributions of this project include:

- Development of a reproducible deep learning pipeline for image classification.
- Evaluation of multiple CNN architectures and their performance.
- Analysis of the model's scalability and training behavior on increasing dataset sizes.

All code is available on GitHub, and a Colab notebook is provided to ensure reproducibility.

## 2 Dataset Description

### 2.1 Origin and Structure of the Dataset

The project uses two publicly available datasets from Kaggle:

- **Amazon Books Dataset** (`mohamedbakhet/amazon-books-dataset`): contains structured metadata for a large number of books, including title, author, publisher, and image URL.
- **Amazon Books Reviews** (`mohamedbakhet/amazon-books-reviews`): provides additional rating-related metadata but was not directly used in the classification task.

Both datasets are licensed under CC0 and were downloaded using the official Kaggle API within the Colab notebook, following the authentication procedure required by the course.

The dataset of interest includes metadata fields such as:

- **Title**
- **Author**
- **Publisher**

- **Published Date**
- **Description**
- **Categories** (list of genres)
- **Image** (URL of the book cover)
- **Info Link, Preview Link**

The most relevant column for this project is `image`, which contains the cover image URL. These covers were downloaded and processed to serve as input to a Deep Neural Network (DNN), with the goal of classifying the books by genre based solely on the visual characteristics of the cover. This defines a pure image classification task.

## 2.2 Sampling Strategy and Dataset Size

The original dataset contains over 2 million rows. For the purpose of this project — which prioritizes scalable design while remaining computationally feasible — a structured sub-sample was created.

A Python preprocessing script was used to:

- Extract the top 3 most frequent book categories.
- Randomly sample 333 books per category using pandas.
- Combine the samples into a new balanced dataset of 999 rows.
- Save the resulting dataset to `dataset/books_subdata.csv`.

The selected categories were:

- `Fiction` (333 books)
- `Religion` (333 books)
- `History` (333 books)

This approach ensured that:

- Each class was balanced for effective training and evaluation.
- The dataset size remained compatible with resource constraints.
- No duplicate image URLs were retained.

### 2.3 Image Download and Preparation

All book cover images were downloaded using a custom MapReduce-style pipeline implemented in Python using the `aiohttp` and `asyncio` libraries. Each image was:

- Fetched from its URL asynchronously.
- Converted to RGB and resized to $224 \times 224$ pixels.
- Stored locally in the `book_covers/` directory.

Only successfully downloaded and valid images were retained in the final dataset. Failed or missing images were filtered out before training.

## 3 Data Organization

### 3.1 Folder Structure and Data Flow

The project maintains a clearly modular structure that separates raw input data, image downloads, intermediate metadata, and final model-ready datasets. The folder and file organization is designed to facilitate reproducibility and scalability:

- `dataset/books_subdata.csv`: Filtered input CSV file containing book metadata and image URLs.
- `book_covers/`: Directory where all downloaded book cover images are stored after validation.
- `books_with_covers.csv`: Intermediate file associating metadata with the successfully downloaded images.
- `books_with_covers_optimized.csv`: Final dataset including image paths and encoded labels, ready for model input.
- `dataset/train_data.csv`, `val_data.csv`, `test_data.csv`: Stratified dataset splits used in training and evaluation.

Each stage of the pipeline builds on the previous one, enabling independent caching and rerun of specific steps.

### 3.2 Image Download and Caching

Images are downloaded using an asynchronous, chunk-based process that supports scalability and fault tolerance. The pipeline leverages `aiohttp` and `asyncio` to process the input CSV in chunks, allowing memory-efficient execution on large datasets.

Each image is:

- Fetched via HTTP from the URL specified in the `image` column.
- Checked for validity and format.
- Saved with a sanitized filename to the `book_covers/` directory.

The images are also converted to RGB format and resized to 224x224 pixels, so that they are immediately compatible with the input data.

This method ensures that successfully processed images are cached locally, avoiding redundant network calls in subsequent runs.

### 3.3 Dataset Splitting

Once images and labels are finalized, the dataset is split into three subsets for training, validation, and testing. The split is performed as follows:

- 70% training set
- 15% validation set
- 15% test set

Splits are generated using stratified sampling based on label frequency to ensure class balance. A fixed random seed (`random_state=42`) is applied to maintain reproducibility across experiments. Labels with fewer than three total samples are filtered out prior to splitting to avoid underrepresented classes.

This threshold was introduced to ensure that every class is represented in all subsets and to avoid rare classes negatively affecting validation or testing. The split is performed in two stages: a first split between training and a temporary block containing validation and testing, followed by a second proportional split between validation and testing. The method keeps the distribution of classes unchanged between the three sets, reducing the risk of learning bias.

## 4 Pre-processing Techniques

### 4.1 Image Preprocessing

To ensure consistency and compatibility with convolutional neural networks (CNNs), all book cover images are preprocessed through two key operations:

- **Resizing:** Each image is resized to a fixed resolution of $224 \times 224$ pixels using the `PIL` library immediately after download. This standardization ensures that all inputs conform to the dimensional requirements of typical CNN architectures.
- **Normalization:** In the TensorFlow data pipeline, pixel values are converted from 8-bit integers ($[0, 255]$) to 32-bit floating-point values in the range $[0.0, 1.0]$ via `tf.image.convert_image_dtype`. This step improves numerical stability and accelerates model convergence.

These transformations are applied consistently across all subsets (training, validation, and test) to maintain a uniform data distribution.

During the pre-processing phase, any corrupted or unreadable images are also filtered out, preventing errors in the training phase.

## 4.2 Data Augmentation

To enhance model generalization and reduce the risk of overfitting, data augmentation is applied dynamically during training using TensorFlow's `tf.keras.Sequential` API. The augmentation pipeline includes the following transformations:

- **Random horizontal flipping**
- **Random rotation** (up to $\pm 10\%$)
- **Random zoom** (up to $\pm 10\%$)

These stochastic augmentations introduce visual variability while preserving the semantic content of the images. Augmentation is applied only to the training set, ensuring a clean validation and test evaluation.

The chosen transformations are light and visually plausible, so as not to alter the semantic content of the cover, but to increase the visual variety relevant for classification.

## 4.3 Label Encoding

Each book in the dataset is associated with a list of categorical tags. For classification purposes, only the primary (first) category is retained. The encoding procedure involves:

- Parsing the `categories` field using `ast.literal_eval` to extract the first entry from the list.
- Converting the extracted main category into a numerical class index using pandas' `category` type:

  ```
  df['label'] = df['main_category'].astype('category').cat.codes
  ```

This process yields a compact integer encoding compatible with TensorFlow's `sparse_categorical_crossentropy` loss function. The result is a clean and consistent label space suitable for supervised multi-class classification.

The approach allows automatic and efficient coding even in the presence of numerous classes, while maintaining a stable and reusable mapping for the inference phase.

## 5 Algorithm and Implementation

### 5.1 Model Architectures

Two different architectures were implemented and evaluated for the task of image-based book classification:

- **Custom CNN:** A lightweight convolutional neural network built from scratch, consisting of three convolutional blocks (`Conv2D` + `MaxPooling2D` + `Dropout`), followed by a fully connected dense layer and a final softmax classification layer. Each block uses increasingly deep filters to progressively capture more complex visual patterns, with dropouts to reduce overfitting.

- **Transfer Learning with MobileNetV2:** A pre-trained MobileNetV2 model (without the top layer) used as a frozen feature extractor. A global average pooling layer and a dense classification head are appended on top.

Both models use an input shape of $224 \times 224 \times 3$, compatible with the preprocessed image resolution. The normalised and rescaled input ensures consistency and performance in standard convolutional models.

### 5.2 Architecture Selection Rationale

The custom CNN is computationally lightweight and suitable for fast experimentation, especially on limited hardware. It enables faster training and easier debugging, making it a good baseline. It was useful to quickly test the preprocessing pipelines and assess the quality of the dataset before investing in more complex models.

The MobileNetV2-based architecture leverages pre-trained features from ImageNet, which significantly improves generalization despite a limited training dataset. It was chosen as a scalable, production-friendly alternative with better performance when additional resources are available. Thanks to the knowledge transfer from ImageNet, it was more stable and accurate from the earliest training periods.

This dual-architecture approach allows flexibility in deployment and experimentation under varying resource constraints.

To allow clarity and scalability, the code is organised in modular scripts. The main functions and their role within the project are listed below.

- `download_dataset(...)`: This function downloads and extracts the dataset from Kaggle, ensuring the required files are locally available.

- `mapreduce_process(...)`: This function performs the download of book covers by splitting the dataset into several chunks. The structure is inspired by the MapReduce paradigm, calling `process_chunk(...)` for each chunk and aggregating the results using `reduce_aggregate_results(...)`.

- `process_chunk(...)` and `map_download_image(...)`: These functions handle asynchronous downloading and image preprocessing using libraries such as `aiohttp` and `asyncio`.
- `enrich_with_labels(...)`: At this stage, the main category is extracted from the `categories` field and encoded as a numeric label. The resulting CSV file is used for training the model.
- `create_dataset(...)`: This function loads images and labels, applies resizing and normalization, and optionally performs data augmentation.
- `split_dataset(...)`: This function splits the labeled dataset into training, validation, and test sets, while preserving the class distribution through stratified sampling.
- `get_model(...)`: Returns the appropriate model architecture based on the selected configuration. Supports both the custom CNN and the MobileNetV2-based model.
- `train_model(...)`: Compiles and trains the model using the Adam optimizer and early stopping callback. The best-performing model is automatically saved using `ModelCheckpoint`.
- `evaluate_model(...)`: This function evaluates the trained model on the test set and, optionally, prints classification metrics.
- `plot_training_history(...)`: Displays the accuracy and loss curves for both training and validation sets using Matplotlib.

## 5.3 Training Configuration

All models were trained using TensorFlow and Keras with the following hyperparameters and callbacks:

- **Loss function:** `sparse_categorical_crossentropy`, compatible with integer-encoded labels.
- **Optimizer:** `Adam`, using the default learning rate (`0.001`) - It is an adaptive optimiser widely used for its efficiency and stability, especially on noisy or unbalanced data.
- **Batch size:** 32. It indicates how many images are processed simultaneously before updating the model weights.
- **Epochs:** Up to 20, with early stopping triggered after 3 epochs of no improvement in validation loss. One epoch corresponds to a complete pass over the entire dataset. The maximum number is limited to 20, but training can stop earlier if there is no improvement.

- **Callbacks:** `EarlyStopping` (patience = 3) and `ModelCheckpoint` to save the best model (`best_model.keras`) based on validation loss.

Model training history (accuracy and loss curves) is visualized using Matplotlib.

### 5.4 Tools and Environment

The project is developed entirely in Python using the following libraries and frameworks:

- `TensorFlow` and `Keras` for model building, training, and evaluation.
- `Matplotlib` for plotting training metrics.
- `Scikit-learn` for dataset stratification and metrics.
- `Google Colab` as the execution environment, leveraging free GPU access for faster training and prototyping.

All code is modular and portable, enabling reproducibility and scalability to larger datasets or more complex architectures with minimal refactoring.

Each component - preprocessing, training, downloading - is separate and reusable, facilitating extension of the project or adaptation to other datasets.

## 6 Scalability of the Solution

### 6.1 Handling Larger Datasets

The solution is engineered to scale efficiently with larger datasets, leveraging chunked processing, asynchronous I/O, and streaming pipelines. Key components include:

- **Chunked Processing:** The CSV file is read in chunks using pandas' `read_csv(..., chunksize=...)` to manage memory efficiently. Each chunk is processed independently during image download via a MapReduce-style loop.
- **Asynchronous Image Downloading:** Using Python's `aiohttp` and `asyncio` libraries, image URLs are fetched in parallel with retry logic and error handling. This non-blocking strategy accelerates data collection and avoids network bottlenecks. Asynchronous downloading makes maximum use of the available bandwidth and minimises the waiting time associated with HTTP requests.
- **TensorFlow `tf.data` Pipeline:** The model input pipeline is implemented using the `tf.data` API, with features such as `map`, `batch`, `prefetch`, and `AUTOTUNE`. These optimizations enable parallel preprocessing and maximize GPU throughput without loading the full dataset into memory.

These mechanisms make the pipeline scalable and adaptable for datasets with tens or hundreds of thousands of images.

### 6.2 MapReduce-Inspired Download Pipeline

To ensure scalability during data acquisition, the image downloading pipeline adopts a structure inspired by the MapReduce programming model:

- **Map Phase:** The dataset is read in manageable chunks using pandas, and each chunk is asynchronously processed via `process_chunk(...)`. This phase downloads and preprocesses book cover images concurrently using `aiohttp` and `asyncio`, enabling parallel, non-blocking execution. Each image is validated, converted to RGB and resized to $224 * 224$ pixels immediately after download, ensuring consistency in the data and minimising subsequent preprocessing steps.
- **Reduce Phase:** The results of each chunk (indexed image paths) are aggregated and integrated back into a central CSV file via the `reduce_aggregate_results(...)` function. This step produces a clean dataset linking metadata to valid local image files.

This modular structure allows the pipeline to scale horizontally. In future extensions, chunks could be distributed across multiple nodes or containerized workers in a cloud environment, enabling massive-scale data ingestion. The independent phase structure also facilitates debugging, maintenance and modular replacement of individual flow components.

### 6.3 Potential Bottlenecks and Mitigation Strategies

Several scalability challenges were considered and addressed:

- **Network Latency and Failures:** Network issues during image fetching are mitigated via retry logic (up to 3 attempts) and selective saving of successfully downloaded images.
- **Disk I/O:** Large-scale datasets may cause disk read/write delays, especially when storing individual image files. Mitigations include chunked writing, optional SSD storage, and decoupling image preprocessing from model training.
- **Memory Constraints:** The modularity of the pipeline allows tuning of `batch_size` and `max_images` parameters to fit the workload to available RAM or GPU VRAM.

### 6.4 Model Complexity and Inference Time

Two models are implemented with different complexity-performance tradeoffs:

- **Custom CNN:** A lightweight sequential model with three convolutional layers, suitable for rapid training and deployment on edge devices or low-resource environments. It offers fast inference times with moderate accuracy. The model consists of about 200,000 parameters, and can complete the inference on a single image in a few CPU milliseconds.

- **MobileNetV2:** A pre-trained deep architecture used in a frozen configuration for transfer learning. It offers higher accuracy on small datasets but has longer inference time and increased memory usage, making it more suitable for server-side inference.

Thanks to the modular design (`get_model(...)`), switching between models or extending to more complex architectures (e.g., ResNet, EfficientNet) requires minimal changes to the code. This approach facilitates comparison between models and adaptation to different computational constraints without changing the preprocessing or evaluation logic.

## 7 Results and Discussion

### 7.1 Custom CNN Performance

The custom convolutional neural network was trained over 10 epochs using a batch size of 32. As shown in the training logs and accuracy/loss plots (Figures **??** and **??**), the model reached a peak validation accuracy of **41.78%**, while training accuracy stabilized around **38.35%**.

On the test set, the final accuracy was **36%**. The model's performance varied across classes, with a precision/recall imbalance that highlights uneven classification capacity (Table 1).

### 7.2 Effect of Dataset Size and Class Balance

The dataset was intentionally limited to 999 samples (333 per class) to simulate a scalable pipeline on a constrained machine. While this allowed architectural testing and pipeline validation, it also introduced limitations in model generalization. The small number of samples per class restricted the network's ability to learn deep visual patterns, leading to moderate performance even after regularization.

### 7.3 Error Analysis

The confusion matrix and classification report indicate that:

- Class 1 was classified with relatively high recall (**76%**), suggesting the model found distinguishing features.

- Classes 0 and 2 suffered from both low recall (**17%** and **12%**, respectively) and low f1-scores.

This disparity may stem from intra-class visual variability, class imbalance in the source dataset, or label ambiguity.

### 7.4 Generalization and Overfitting

The training and validation accuracy curves show signs of minor overfitting after epoch 6, where training accuracy rises but validation performance fluctuates. However, the use of dropout layers and early stopping helped mitigate this.

Given the small dataset, the model maintained relatively good generalization without severe divergence. With more data and finer augmentation, the model would likely improve in both stability and accuracy.

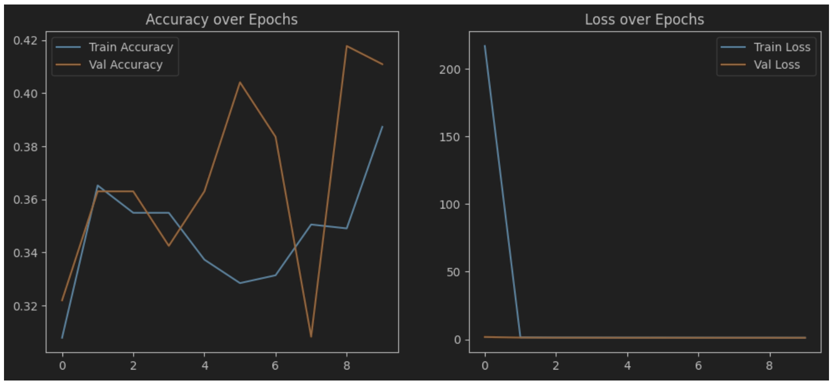

Figure 1: Training and validation accuracy and loss over 10 epochs for the custom CNN.

| Class | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| 0 | 0.33 | 0.17 | 0.22 | 48 |
| 1 | 0.35 | 0.76 | 0.48 | 50 |
| 2 | 0.46 | 0.12 | 0.20 | 48 |

Table 1: Classification report for the custom CNN model on the test set

### 7.5 MobileNetV2 Performance

The MobileNetV2 model was trained for 10 epochs using frozen pretrained weights from ImageNet. As shown in Figure 2, the model reached a validation accuracy of **46.58%** and a training accuracy of **54.20%** in the final epoch. This represents a notable improvement over the baseline CNN, particularly in terms of stability and generalization.

On the test set, the final accuracy was **43.84%**, with nearly balanced precision, recall, and F1-scores across all three classes (see Table 2).

### 7.6 Generalization and Learning Behavior

The MobileNetV2 architecture demonstrated superior generalization, reflected in the smoother convergence of both loss and accuracy across epochs. The gap between training and validation performance remained moderate, indicating good regularization even without fine-tuning the base layers.

The pretrained backbone likely enabled more robust feature extraction even from limited training data, which helped mitigate overfitting.

### 7.7 Per-Class Performance

The classification report highlights:

- More balanced F1-scores across all three classes (≈ 0.41–0.48).
- Significant improvement in **Class 0 and 2** recall compared to the custom CNN.
- No single class dominates prediction quality, suggesting improved representation learning.

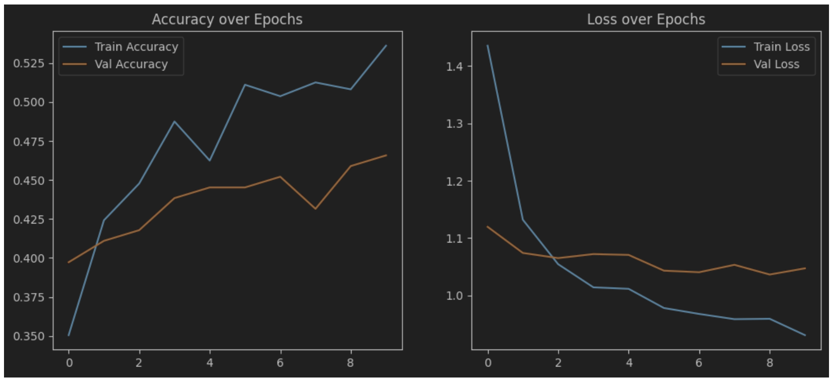

Figure 2: Training and validation accuracy and loss for MobileNetV2 over 10 epochs.

| Class | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| 0 | 0.46 | 0.50 | 0.48 | 48 |
| 1 | 0.42 | 0.42 | 0.42 | 50 |
| 2 | 0.43 | 0.40 | 0.41 | 48 |

Table 2: Classification report for MobileNetV2 on the test set.

## 8 Conclusion

This project addressed the problem of automatic classification of book covers from Amazon using deep learning techniques for image analysis. Two convolutional neural network architectures were implemented and compared: a custom CNN designed from scratch and a MobileNetV2 model based on pretrained weights. The entire processing pipeline — from dataset construction and asynchronous image acquisition to preprocessing, training, and evaluation — was built with modularity and scalability as core design principles, in line with the goals of massive data analysis.

The custom CNN demonstrated the effectiveness of lightweight architectures for rapid experimentation and environments with limited computational resources. It achieved reasonable accuracy and fast inference times, making it suitable as a deployable baseline model. On the other hand, MobileNetV2 delivered significantly improved performance across all classes, benefiting from transfer learning and robust feature extraction. However, it required more memory and longer inference times, highlighting the trade-off between accuracy and computational cost.

Several lessons emerged from this project. First, data preprocessing and label encoding play a crucial role in stabilizing training when working with real-world, noisy datasets. Second, the use of stratified sampling and augmentation proved valuable for generalization, even in low-data regimes. Finally, the pipeline's chunk-based, asynchronous MapReduce-inspired architecture offers a practical foundation for scaling to much larger datasets in future work.

Potential directions for extension include:

- Incorporating multimodal data by combining visual features with textual metadata such as title, description, and reviews.
- Fine-tuning pretrained architectures instead of freezing them, to further improve accuracy.
- Deploying the pipeline in a distributed or cloud-based environment to enable full-scale processing on real Amazon catalog data.

Overall, the project demonstrates how scalable and reproducible machine learning workflows can be built even under resource constraints — a critical aspect of applied massive data analysis.

## Appendix

- GitHub Repository
- Colab Notebook Link