COMP5329 Deep Learning Assignment 2

Group members:
Tianzhijie Chen (540349603),
Poon Ka Nok Anson (540349681),
Zhaoting Yang (530698180)

21/5/2025

1 Introduction

1.1 Goal and Importance of Study

The objective of this assignment is to conduct a multi-label image classification competition with Kaggle. We aim to develop deep neural networks with CNN architectures and text encoder to classify each image into category instances. Our models will be trained using the training data set and predicted with the no-label testing set. The results will be submitted to kaggle to obtain a check of the accuracy of the average F1 score.

Multi-Label Classification is a key problem in the field of deep learning, which centers on assigning a set of relevant labels to each sample rather than just a single label. This approach has a wide range of real-world applications, including areas such as medical diagnosis, sentiment analysis, music classification, and image/video annotation (Tarekegn, Giacobini, and Michalak, 2021). Unlike traditional single-label classification methods, multi-label classification is able to handle situations where an instance may belong to more than one category, which puts higher demands on the design of machine learning algorithms .

1.2 Overview of Techniques

In this assignment, various techniques were employed to achieve robust performance on the dataset of 40,000 images with 19 labels (1-19, excluding 12). CNN models such as Inception, VGG, and EfficientNet-B4 are utilized for image feature extraction, with EfficientNet-B4 ultimately chosen for its balance of efficiency and accuracy, complemented by BERT for encoding captions to leverage textual context. The model was trained using Focal Loss with label smoothing to address class imbalance, optimizing for hard-to-classify examples.Evaluation was performed using the Micro F1-Score, with threshold optimization to maximize performance.

1.3 Dataset

The provided data for this multi-label classification task contains 40,000 JPG images with a single or several labels and a short caption explaining the contents of the image. The labels range from 1 to 19, but without label 12, as it does not exist in the dataset. The images are split into training and testing sets, with label information provided for the training set

through accompanying metadata files. The captions provide a text description of the images' content, typically encapsulating actions or objects in the image, that provides additional contextual information that may further contribute to model performance during training and prediction. One of the challenging tasks in this dataset is label imbalance, which can

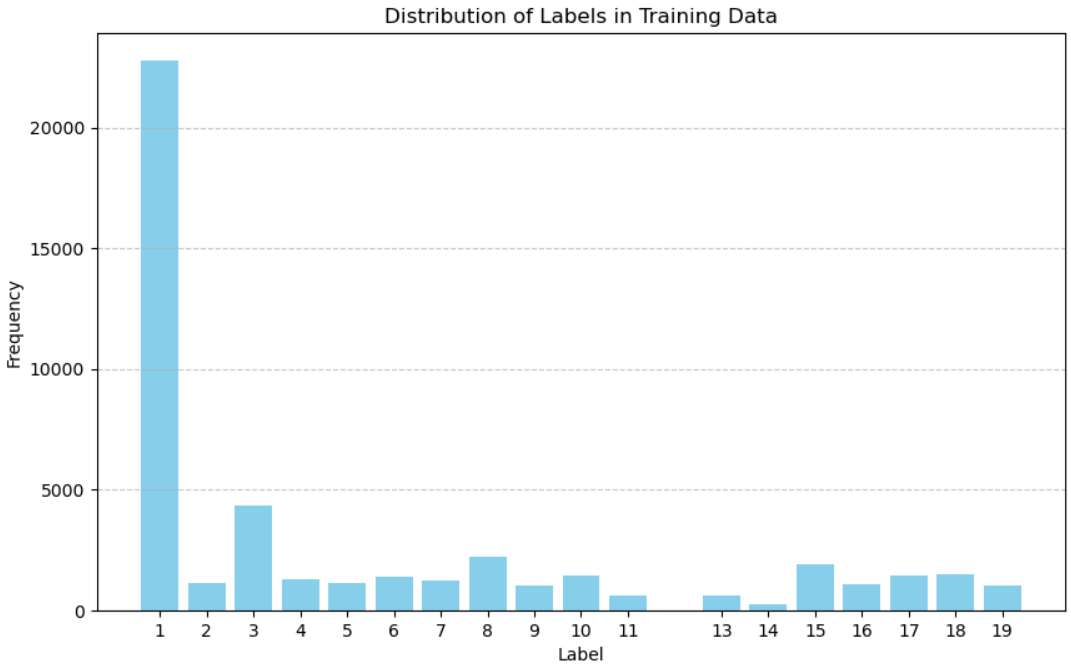

Figure 1: Enter Caption

be observed from the bar plot below. Label 1 is the dominant label in the dataset, and its frequency is over 20,000 occurrences, while other labels, i.e., labels 2 through 19, have relatively low occurrences, in the range of about 1,000 to 3,000 instances. This imbalance poses a challenge during the training of deep learning models as the abundance of label 1 may overshadow the representation and predictability of underrepresented labels.

1.4 Previous work

In early research on multi-label classification (MLC), Tarekegn et al.[1] identified two main approaches to solving the MLC problem: problem transformation and algorithmic adaptation. By transforming a multi-label task into multiple single-label classification tasks or label ranking tasks to solve the MLC problem, the article delved into the applications and limitations of methods such as binary relevance (BR), labeled power set (LP), and classification chaining (CC). To further improve the MLC performance, the researcher puts a hybrid approach of CNN and BERT models.CNN performs well in image classification task and BERT performs well in processing text data. By combining CNN and BERT, it enables these multimodal methods to extract complementary information from both images and text, which significantly improves the prediction accuracy of multi-label classification tasks.

In addition Tarekegn gives a comprehensive review of the imbalance problem in multi-label classification, he points out that imbalanced datasets are a major challenge in MLC, especially when the number of samples for some labels is much lower than the others, the prediction of the model can be seriously affected. Various methods such as resampling methods, classifier adaptation methods, and integration methods are introduced in the study for mitigating the problems associated with label imbalance. When confronted with unbalanced data, the study by Bogatinovski et al. (2022)[2] provides an in-depth comparative analysis of MLC methods using 20 different performance evaluation criteria. The results of the study show that Random Forest Predictive Clustering Trees (RFPCT), Random Forests based on Decision Trees (RFDTBR) and AdaBoost.MH are some of the top performing methods. To get better label prediction, neural network architecture plays a key role in the model. Transnetwork models need to perform effective feature extraction to improve accuracy while managing computational efficiency. Some architectures such as AlexNet, ResNet, and DenseNet, have been successfully applied to multi-category classification tasks, and the results show that Enet and GoogleNet outperform in terms of accuracy per parameter, which suggests that the architecture choice is also particularly important to improve the performance of MLC tasks.

2 Methods

2.1 Data Preprocessing

Large-scale data preprocessing was done to acquire images and labels into their respective formats for training and testing multi-label classification models. Various transformations were employed to increase the training data, enhance model generalization, and offer consistency for validation and testing processes. The below are the data preprocessing steps explained in the subsequent subsections according to image transformations, loading the data, labeling processing, and datasets preparation for PyTorch.

2.1.1 List of Augmentation and Transformation Techniques

The following techniques were applied to augment and transform the training images, ensuring improved model generalization and robustness against overfitting:

- **RandomResizedCrop**: This technique randomly crops a portion of the image and resizes it to a fixed resolution of 224×224 pixels, with antialiasing enabled (`size=(224, 224), antialias=True`). The random cropping introduces variability in the spatial focus of the image, allowing the model to learn features from different regions, while the resizing ensures consistent input dimensions for the convolutional neural network. Antialiasing helps reduce artifacts during resizing, preserving image quality for better feature extraction.
- **RandomHorizontalFlip**: Images are flipped horizontally with a probability of 50% (`p=0.5`). This augmentation simulates natural variations in object orientation, such as a person facing left or right, enhancing the model's ability to generalize across mirrored

perspectives. It is particularly effective for datasets where horizontal symmetry is common, ensuring the model does not overfit to a specific orientation.

- **RandomRotation**: This transformation rotates images by a random angle within the range of ±20 degrees (`degrees=20`). By introducing rotational variability, the model learns to recognize objects regardless of their orientation, which is crucial for real-world applications where objects may appear at various angles. The limited range of rotation ensures that the transformations remain realistic and do not distort the image excessively.

- **ColorJitter**: The brightness, contrast, and saturation of images are randomly adjusted by up to ±30% (`brightness=0.3, contrast=0.3, saturation=0.3`). This technique simulates diverse lighting conditions and color variations that may occur in real-world scenarios, such as changes in illumination or camera settings. By training on these variations, the model becomes more robust to environmental factors, improving its performance on unseen data.

- **RandomAffine**: This transformation applies random translations of up to ±10% of the image dimensions (`translate=(0.1, 0.1)`) and scales the image between 80% and 120% of its original size (`scale=(0.8, 1.2)`), with no rotation applied (`degrees=0`). The translation shifts the image content within the frame, while scaling adjusts the object size, mimicking variations in object distance or camera zoom. These geometric transformations help the model generalize to different spatial configurations without altering the orientation of the content.

- **RandomErasing**: A random rectangular patch, covering between 2% and 20% of the image area (`scale=(0.02, 0.2)`), is erased with a 30% probability (`p=0.3`). This technique replaces the patch with random noise, forcing the model to focus on other regions of the image for feature extraction. It acts as a form of regularization, reducing the risk of overfitting by preventing the model from relying too heavily on specific parts of the image, thus improving its robustness to occlusions.

- **ToDtype**: Images are converted to the `float32` data type and their pixel values are scaled to the range $[0, 1]$ (`dtype=torch.float32, scale=True`). This transformation standardizes the image data for subsequent normalization and ensures compatibility with PyTorch's tensor operations. Scaling to $[0, 1]$ normalizes the pixel intensity range, which is a prerequisite for applying mean and standard deviation-based normalization techniques.

- **Normalize**: ImageNet normalization is applied, subtracting a mean of $[0.485, 0.456, 0.406]$ and dividing by a standard deviation of $[0.229, 0.224, 0.225]$ for each RGB channel, respectively. This process centers the image data around zero with a unit variance, aligning the input distribution with the statistics of the ImageNet dataset, on which many pretrained models are based. This normalization ensures that the model's convolutional layers receive inputs in a standardized format, facilitating faster convergence and better feature extraction.

Figure 2: Example of Transformed Image

2.1.2 Test-Time Transformations

For the validation and test sets, a minimal set of transformations was applied using the `test_transforms` pipeline in the `MultiLabelClassifier` and `MultiLabelPredictor` classes to ensure consistent input formatting without introducing randomness. The process begins with `Resize`, which resizes images to 224×224 pixels (`size=(224, 224), antialias=True`) to match the training input dimensions. This is followed by `ToDtype`, which converts images to `float32` and scales them to the range $[0, 1]$ (`dtype=torch.float32, scale=True`). The final step applies the same ImageNet normalization as in the training set, using a mean of $[0.485, 0.456, 0.406]$ and standard deviation of $[0.229, 0.224, 0.225]$ per channel, ensuring that the validation and test inputs are processed consistently with the training data.

2.1.3 Loading and Validation

Several validation checks were implemented to ensure reliable data input during the loading process. The presence of essential files, including `train_cleaned.csv`, `test_cleaned.csv`, and the `data/` folder containing the images, was verified to avoid runtime errors. Only `.jpg` files were processed, with their existence confirmed prior to loading. Images were loaded using the PIL library, converted to RGB mode to ensure consistency across the dataset, and resized to 224×224 pixels. To handle potential issues, exception handling was incorporated to skip corrupted images or invalid labels, thereby preventing crashes during training or evaluation and ensuring a stable data pipeline.

2.1.4 Label Processing

The multi-label annotations were carefully processed to prepare them for model training. Labels were extracted from space-delimited entries in the metadata files, with invalid entries—non-numeric values, label 12, or labels outside the range 1–19—filtered out to maintain data integrity. The processed labels were then converted into one-hot vectors for 18 classes (labels 1–11, 13–19) using scikit-learn's `MultiLabelBinarizer`, enabling the model to handle multi-label classification effectively. Additionally, label frequencies were computed to determine the α parameter in the Focal Loss function, which balances the contribution of

each class during training and addresses label imbalance in the dataset, ensuring appropriate weight for underrepresented classes.

2.1.5 Preparing the Dataset

The dataset was formatted for efficient use with PyTorch to facilitate training and evaluation. Images, initially loaded as NumPy arrays, were converted to PyTorch tensors and permuted from HWC to CHW format (`torch.from_numpy(img).permute(2, 0, 1)`), as required by PyTorch's convolutional layers. The training data was split into 80% training and 20% validation sets with random shuffling (`train_split=0.8, np.random.seed(42)`) to ensure reproducibility of the split. PyTorch `DataLoader` objects were created with a batch size of 16 (`batch_size=16`); the training loader used shuffling (`shuffle=True`), while the validation and test loaders did not (`shuffle=False`). Additional settings, including `num_workers=0` and `pin_memory=False`, were applied to ensure compatibility with the MPS device, enabling efficient data loading and batch processing during model training and evaluation.

2.2 Loss Function

To address the challenges of multi-label classification, particularly the class imbalance observed in the dataset, a Focal Loss with Label Smoothing was employed as the primary loss function. This loss function was implemented in the `FocalLoss` class (Cell 3) and instantiated within the `MultiLabelClassifier.setup_model` method (Cell 6) with the following configuration: `self.criterion = FocalLoss(alpha=self.alpha, gamma=2.5, label_smoothing=0.1)`. The α parameter was dynamically computed in `MultiLabelClassifier.__init__` as the mean inverse class frequency, ensuring that rare classes are given higher weight during training. Specifically, the class frequencies were calculated using a `Counter` on the training labels, with α set as the mean of total/(len(label_counts)$\cdot v$) for each class frequency v, where total is the sum of all label occurrences. This approach balances the contribution of each class, addressing the dominance of label 1 in the dataset.

The Focal Loss is calculated from Binary Cross-Entropy (BCE) with logits, computed via `nn.BCEWithLogitsLoss` with `reduction='none'` to determine the base loss for each class. The BCE loss is then modified by a focusing term, $\alpha \cdot (1 - p_t)^\gamma \cdot \text{BCE\_loss}$, where $p_t = \exp(-\text{BCE\_loss})$ represents the probability of the true class, α adjusts for class imbalance, and $\gamma = 2.5$ promotes hard-to-classify samples by assigning less weight to easy ones. A higher γ increases the focus on difficult samples, making the loss more effective for imbalanced datasets. The Focal Loss equation can be expressed as follows:

$$\text{FocalLoss}(x, y) = \alpha \cdot (1 - p_t)^\gamma \cdot \text{BCEWithLogitsLoss}(x, y), \tag{1}$$

where

$$\text{BCEWithLogitsLoss}(x, y) = -[y \cdot \log(\sigma(x)) + (1 - y) \cdot \log(1 - \sigma(x))], \tag{2}$$

$\sigma(x)$ is the sigmoid function, and

$$p_t = \exp(-\text{BCEWithLogitsLoss}(x, y)). \tag{3}$$

Label smoothing was also utilized with label_smoothing=0.1, modifying the target labels as:

$$\text{targets} = \text{targets} \cdot (1 - \text{label\_smoothing}) + \text{label\_smoothing/inputs.size(-1)}. \quad (4)$$

This reduces overconfidence in predictions and improves generalization by preventing the model from being too certain about noisy or uncertain labels. The final loss is averaged across all elements using `reduction='mean'`. During training, the loss is computed for each batch within the `train_and_validate` method as `loss = self.criterion(outputs, labels)`, accumulated over `accum_steps=2`, and backpropagated to update the model weights. This loss function enhances the stability of training for the 18-class task (labels 1–11, 13–19) by focusing on hard examples, appropriately weighting underrepresented classes, and improving robustness to noisy labels through label smoothing, ultimately contributing to improved F1 scores.

2.3 Validation Metrics

Model performance was evaluated using a range of metrics, with F1 score being employed as the primary metric for model selection and early stopping. There were three primary metrics: a multi-label F1 score, an F1 score per class with threshold optimization, and validation loss, all providing complementary information regarding the performance of the model on the validation set.

2.3.1 Multi-Label F1 Score

The multi-label F1 score was computed using `torchmetrics.F1Score` with the configuration `task="multilabel", num_labels=self.num_classes`, where `num_classes=18`, as defined in `MultiLabelClassifier.setup_model`. The metric was moved to the MPS device (`to(self.device)`) for high-speed calculations. In the `train_and_validate` method, the F1 score per batch was calculated as `self.f1(outputs, labels).item()` and averaged across the validation set to obtain the mean F1 score.

This metric computes the harmonic mean of precision and recall across all 18 classes, providing a balanced measure of model performance in the presence of class imbalance. The F1 score is defined as follows, where p is precision (the ratio of true positives to predicted positives), r is recall (the ratio of true positives to actual positives), tp is the number of true positives, fp is the number of false positives, and fn is the number of false negatives:

$$F_1 = 2\frac{p \cdot r}{p + r}, \quad \text{where} \quad p = \frac{tp}{tp + fp}, \quad r = \frac{tp}{tp + fn}. \quad (5)$$

The F1 score served as the primary criterion for early stopping and model selection, with the best F1 score tracked as `best_f1`. If the mean F1 score exceeded the current best F1 score, the model was saved, and the patience counter was reset, as shown below:

```
if mean_f1_score > best_f1:
    best_f1 = mean_f1_score
    patience_counter = 0
    self.save_model(path=self.model_save_path, epoch=epoch)
```

This metric directly aligns with the goal of achieving an F1 score of at least 0.885, making it a critical indicator of model performance.

2.3.2 Threshold-Optimized F1 Score

To further enhance the F1 score, a threshold optimization procedure was implemented in the `optimize_threshold` method. For each of the 18 classes, a range of thresholds from 0.3 to 0.7 (with a step size of 0.05) was tested to maximize the binary F1 score (`task="binary"`) per class. For each class $c \in \{1, 2, \ldots, 18\}$, the optimal threshold θ_c was determined by:

$$\theta_c = \arg \max_{\theta \in [0.3, 0.7]} F_1(\text{pred}_c(\theta), y_c), \tag{6}$$

where $\text{pred}_c(\theta) = \mathbb{I}(\sigma(x_c) > \theta)$ is the predicted binary label for class c, $\sigma(x_c)$ is the sigmoid-transformed output probability, and y_c is the true binary label. The outputs were passed through a sigmoid function (`torch.sigmoid(outputs).cpu().numpy()`), and labels were converted to NumPy arrays for processing. The best thresholds were then applied to compute the overall multi-label F1 score (`task="multilabel", num_labels=self.num_classes`), which assesses performance across all classes simultaneously. This process optimizes the decision boundary for each class, accounting for class-specific confidence levels, which is particularly beneficial for imbalanced classes. This metric provides a key indicator of potential test set performance and is critical for achieving the target F1 score of 0.885 or higher.

2.3.3 Validation Loss

The validation loss was computed using the same Focal Loss criterion as during training, with α based on class frequencies, $\gamma = 2.5$, and label_smoothing=0.1. The Focal Loss is defined as follows:

$$\text{FocalLoss}(x, y) = \alpha \cdot (1 - p_t)^\gamma \cdot \text{BCEWithLogitsLoss}(x, y), \tag{7}$$

where $\text{BCEWithLogitsLoss}(x, y) = -[y \cdot \log(\sigma(x)) + (1-y) \cdot \log(1-\sigma(x))]$, $\sigma(x)$ is the sigmoid function, and $p_t = \exp(-\text{BCEWithLogitsLoss}(x, y))$. In the `train_and_validate` method, the loss was calculated for each batch under automatic mixed precision, as shown below:

```
with torch.amp.autocast(device_type=autocast_device):
    outputs = self.model(images)
    loss = self.criterion(outputs, labels)
```

The total validation loss was accumulated and averaged over the validation set to compute the average validation loss:

$$\text{avg\_val\_loss} = \frac{1}{N} \sum_{i=1}^{N} \text{FocalLoss}(x_i, y_i), \tag{8}$$

where N is the number of batches in the validation set, and $\text{FocalLoss}(x_i, y_i)$ is the loss for the i-th batch. This computation was implemented as:

```
avg_val_loss = total_val_loss / len(self.val_loader)
```

This metric monitors model convergence and the risk of overfitting, complementing the F1 score. The validation loss also drives the learning rate scheduler (`ReduceLROnPlateau`), which adjusts the learning rate based on `avg_val_loss` (`self.scheduler.step(avg_val_loss)`).

3 Convolutional Neural Network Architecture

3.1 CNN

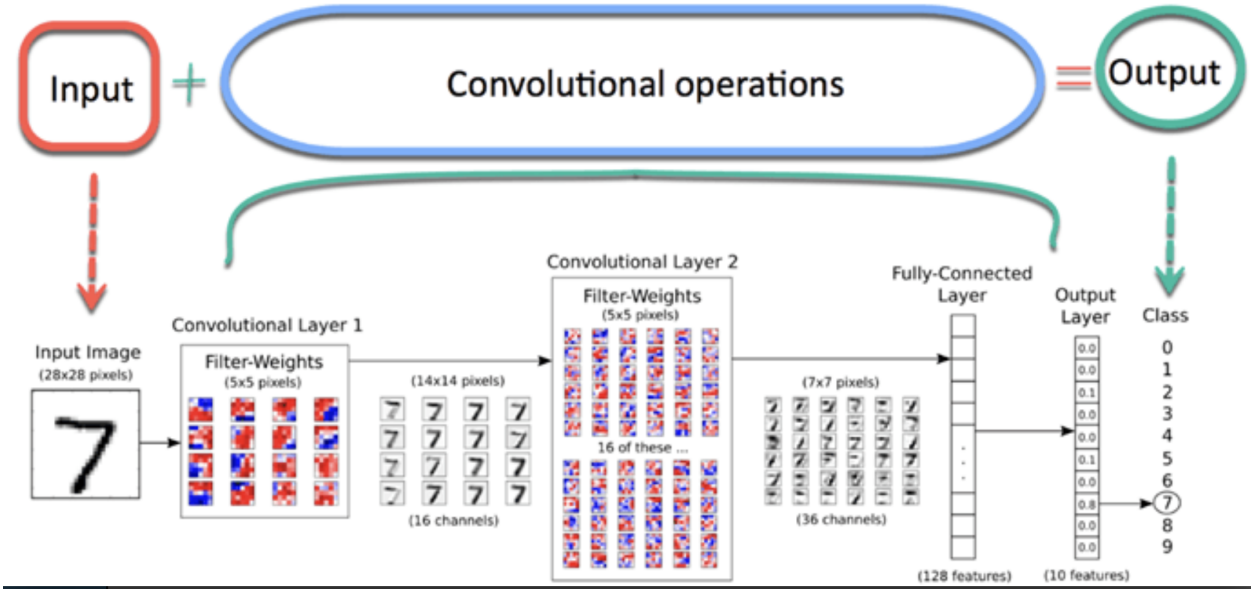

Figure 3: CNN Architecture (Daniel, 2021)

CNN is a neural network structure commonly used in the field of deep learning, which is widely used in computer vision tasks, such as image classification, object detection, etc. CNN extracts useful features from an image by simulating the processing of human vision using convolutional and pooling layers, and then performs the final classification or regression prediction through fully connected layers. The process of CNN involves: first, a convolutional layer slides over the image with a convolutional kernel to extract local features. Then, the pooling layer downsamples the feature map to reduce the size of the image while retaining important features, thus reducing the amount of computation and preventing overfitting. Next, the fully connected layer spreads the convolved and pooled feature maps and feeds them into a neural network that combines the features through learning for the final classification or regression task. Finally, the output layer transforms the results to generate the final prediction.[3]

3.2 ResNet18

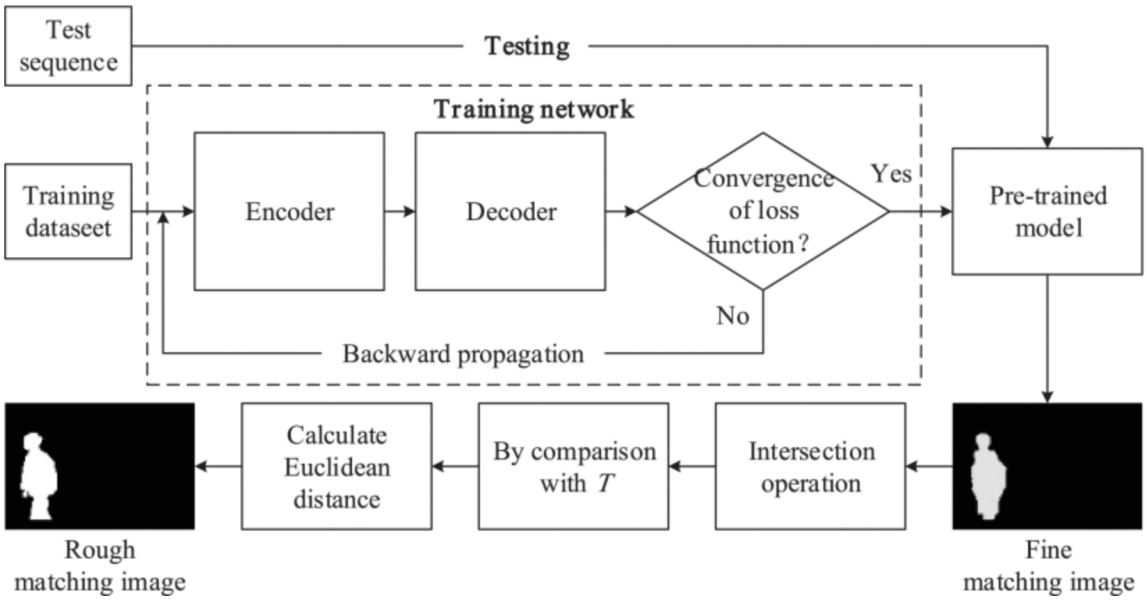

Figure 4: ResNet18 Architecture (Ou et al., 2019)

Residual Neural Network is used for computer vision tasks such as image classification. The innovation of ResNet compared to traditional CNN is its introduction of Residual Block, which solves the problem of gradient vanishing that may be encountered in model training as the number of network layers increases. Through "skip connections" or "shortcut connections," the output of each layer can be directly added to the output of previous layers, helping the gradient to propagate through deeper networks. The specific training process of ResNet18 model is firstly, the input image goes through a convolutional layer for feature extraction, and the residual connection ensures that the gradient is passed effectively and avoids the problem of gradient vanishing in the deep network. Then pooling layer is used for dimensionality reduction and retaining important features, nonlinear transformation is performed by activation function and final output is generated in fully connected layer. The gradient of the loss function is computed using back propagation and the model parameters are updated to reduce the error. Xianfeng Ou et al. (2019) [4]proposed the ResNet18 model and provided a detailed description of the architecture for this figure.The dotted box shows training network, in which training data contains image sequences and corresponding artificial labels. Firstly, they are input into the network for training until the loss function converged. And then pre-trained model will be generated when network training is completed. Finally, finish the prediction of the classification of pixel points and output rough matching images by importing the image sequence into the pre-trained model.

3.3 VGG16

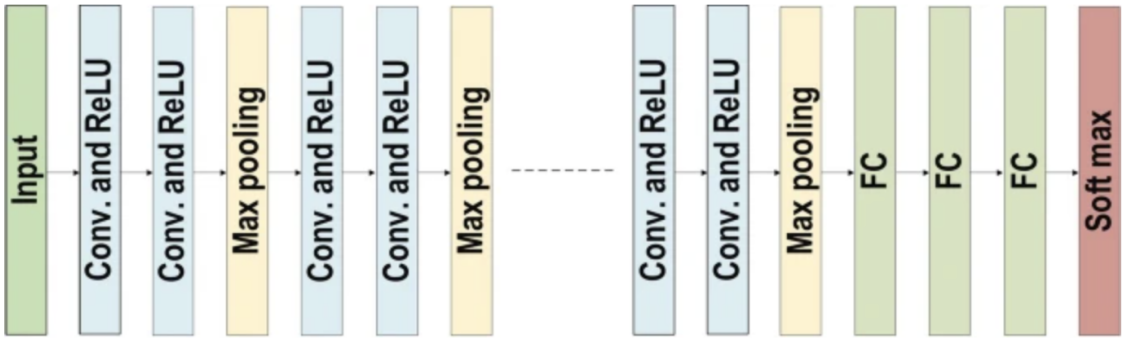

Figure 5: VGG16 Architecture (Alzubaidi et al., 2021)

Simonyan and Zisserman proposed a simple and efficient design principle for CNNs, known as VGG. [5] VGG16 is a deep convolutional neural network architecture designed for image classification tasks. It consists of 16 weight layers, including 13 convolutional layers and 3 fully connected layers. The main feature of this architecture is the use of small 3x3 convolutional filters, which are stacked deeper to capture complex features in the images. The working process of VGG16 consists of processing the input image through multiple convolutional, pooling and fully connected layers. First, the input image is passed through convolutional layers to extract features, using a small 3x3 convolutional kernel to slide the image, gradually capturing simple to complex image features. Each convolutional layer is followed by a ReLU activation function to increase the nonlinearity. The pooling layer then downsamples the convolved feature maps to reduce the spatial dimensions, reduce computation and prevent overfitting. Finally the pooled feature maps are spread and fed into the fully connected layers, through which the extracted features are synthesized to output the classification results of the image. The output layer generates the probability of each category by softmax function and selects the category with the largest probability as the prediction result of the image.

3.4 CNN2

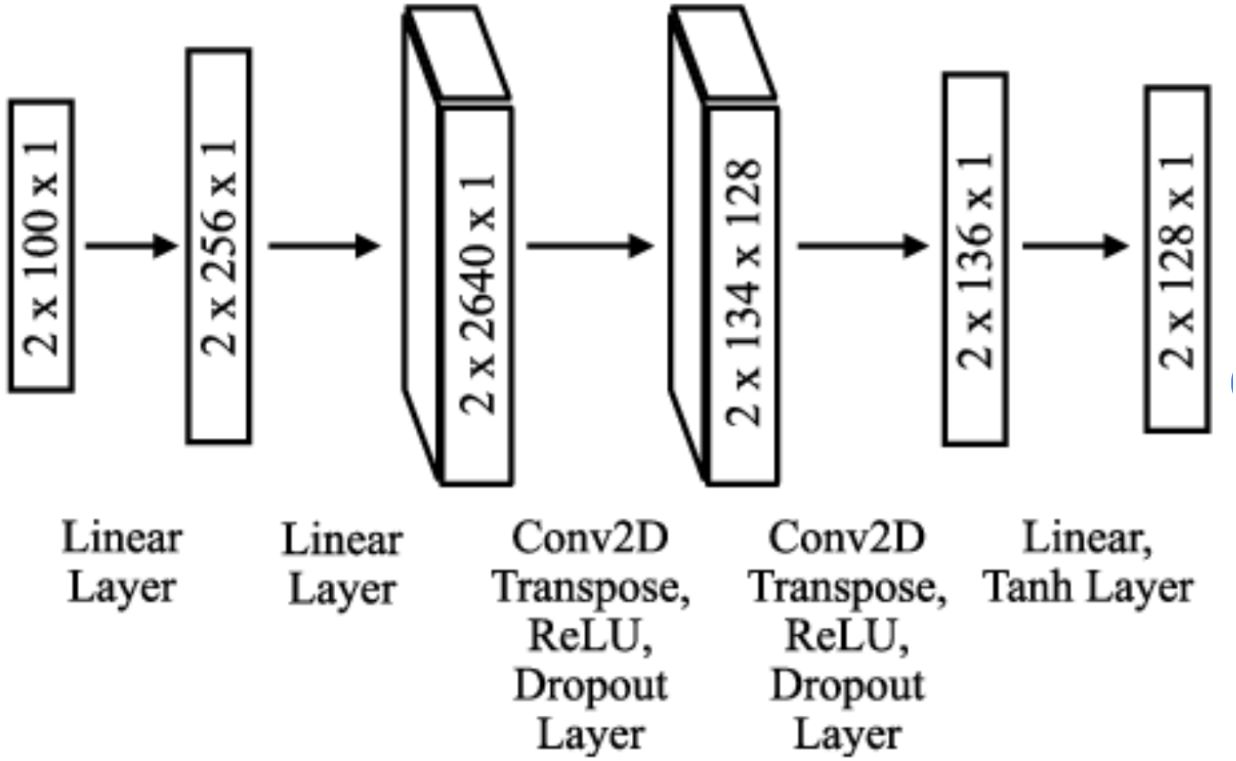

Figure 6: CNN2 Architecture (Freitas et al., 2022)

CNN2 is an improvement of CNN. Compared to traditional CNN, CNN2 has deeper layers, more complex network structure and different convolutional layers and activation functions. [6] In the working process, it is similar to traditional CNN with optimization in hierarchical feature extraction. First, the input image is feature extracted through multiple convolutional layers, each of which uses a convolutional kernel to scan the image to extract features. Compared to traditional CNNs, CNN2 may use smaller convolutional kernels and more convolutional layers to capture details. The output of the convolutional layers is then nonlinearly transformed by an activation function to increase the expressive power of the network. Then the image is flattened and output to the fully connected layer to get the classification result.

3.5 EfficientNet-B4

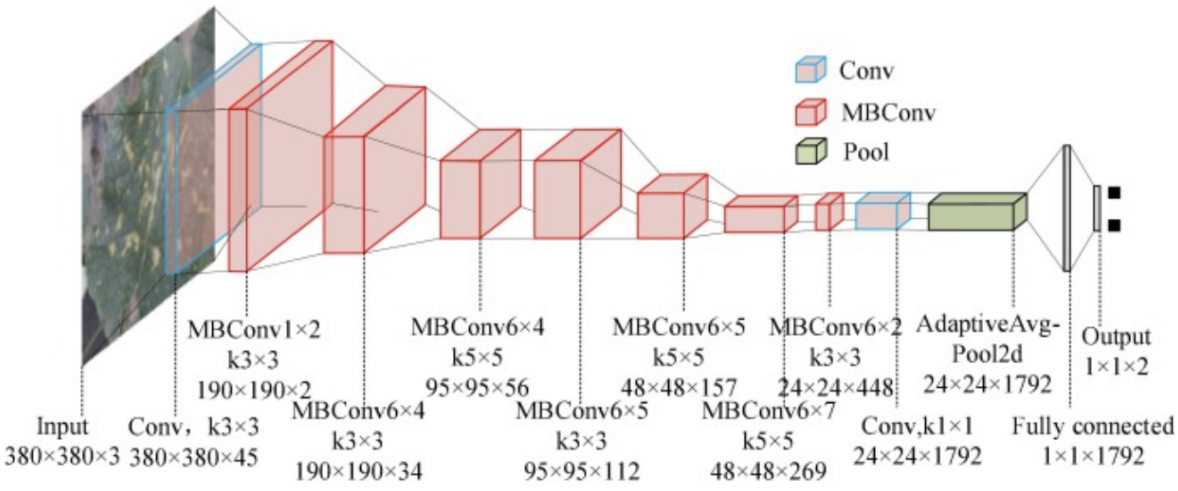

Figure 7: EfficientNet-B4 Architecture (Zhang et al., 2020)

EfficientNet-B4 is an efficient convolutional neural network architecture that achieves high performance in image classification tasks by balancing the model's depth, width, and resolution. Figure 6 [7] shows the authors using the EfficientNet-B4 model to solve the problem of classifying plant diseases. In the case of multiple diseases that may occur on the same leaf, first, when the characteristics of one disease are significantly better than those of another disease, it tends to cause similarity, thus posing a challenge for constructing model accuracy. This model was chosen in the article to construct a binary classification model for diseases with similar characteristics. The working procedure is to first extract the base features by a regular convolutional layer, followed by feature extraction using the Mobile Inverted Bottleneck Convolution (MBConv) layer, which uses depth-separable convolution to extract more complex features while reducing the computational effort. The pooling layer then reduces the size of the feature maps and retains important information by adaptively averaging the pooling to resize the feature maps to a fixed size. Finally, the model generates classification results through a fully connected layer.

4 Experiments

4.1 Model Comparison Analysis

To determine the optimal architecture to obtain this classification function, several neural network architectures were trained and evaluated. The following table compares each model's performance on training, validation, and test sets with Train Loss, Validation Loss, and F1 Score metrics. The model with the highest mean F1 score will be subjected to Ablation Studies to analyze why it performs so well and whether there exists any parameter that could be optimized. We observed that VGG performed the worst among all models with the largest training, validation, and test losses, and a relatively low F1 score. This is likely

due to the fact that VGG utilizes a very deep architecture with small 3x3 filters, and that kind of architecture tends to lead to overfitting or be unable to recognize a range of or complex features in the dataset, especially for small or complex objects. EfficientNetB4 was exceptional in having the smallest training, validation, and test set losses as well as achieving the highest F1 score. Its superior performance is attributed to the compound scaling of EfficientNetB4 in optimizing depth, width, and resolution of the network to provide effective feature extraction and robust generalization. ResNet too performed well, with good F1 score and low losses, owing to its residual links that enable deep networks through avoiding vanishing gradient issues and enhancement of feature propagation.

4.1.1 Key Metrics Evaluation

Table 1: Performance Metrics for Different Models Across Epochs

Model	**Epoch**	**Train Loss**	**Validation Loss**	**F1 Score**
SimpleCNN	1	0.1461	0.0611	0.5531
	2	0.0636	0.0553	0.5531
	3	0.0590	0.0573	0.5418
CNN2	1	0.7548	0.3218	0.5478
	2	0.1644	0.0767	0.5314
	3	0.0718	0.0615	0.5513
ResNet18	1	0.0724	0.3017	0.5351
	2	0.0542	0.0724	0.5423
	3	0.0525	0.0628	0.4864
VGG16	1	0.1624	0.0753	0.5531
	2	0.0638	0.0582	0.5531
	3	0.0556	0.0555	0.5531
EfficientNet-B4	1	0.0735	0.0535	0.5719
	2	0.0410	0.0441	0.6299
	3	0.0335	0.0426	0.6696

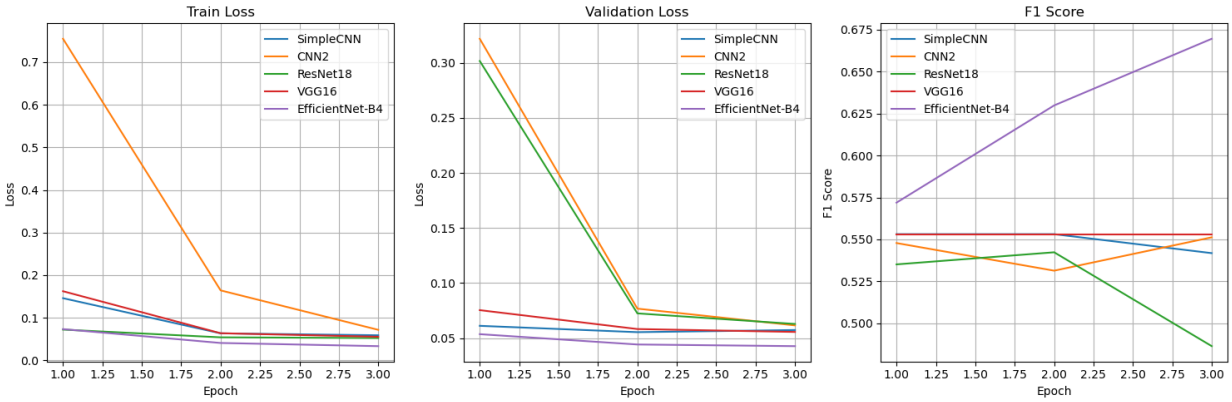

Figure 8: Graphical Key Metrics Comparison

5 Ablation Studies

This section examines the effect of various components and configurations on the performance of the best-performing MultiLabelClassifier model, as determined by the mean F1 metric. The configurations tested include the inclusion or exclusion of two-ward caption encoding, the type of loss function (Focal Loss with label smoothing or BCE Loss), and different learning rates. The findings are presented in Table 2, and the contribution of each component to the model's performance is elaborated below.

5.1 Caption Encoding using two-letter encoding

The two-letter encoding method transforms captions into numerical features by capturing the frequency of two-letter combinations from the lowercase alphabet. it creates a dictionary mapping all possible two-letter pairs (e.g., "aa" to "zz"),totally $26^2 = 676$ combinations, to unique indices.For each caption, it extracts lowercase letters, forms two-letter sequences, and increments the corresponding index in a 676-dimensional vector.

The results show that the model with two-letter encoding achieves a validation mean F1 score of 0.8422, compared to 0.8054 without it. This is a 3.7 percentage point increase, indicating that text incoding contextualized understanding of captions provides valuable additional information for multi-label classification. The validation loss also decreases with encoding (0.0220 compared to 0.0213), suggesting a a better generalization. However, the training loss is slightly higher with two-letter encoding (0.0211 vs. 0.0174), which may indicate the higher complexity of the model when features are derived from encoded captions. These results confirm that two-letter caption encoding significantly enhances the model's performance in predicting correct labels, likely by capturing semantic relationships in captions relevant to the image labels.

5.2 Loss Functions

The loss function was evaluated by comparing Focal Loss with label smoothing (0.1) to BCE Loss (Binary Cross-Entropy Loss). Focal Loss is designed to address class imbalance

by focusing on difficult-to-classify instances, while label smoothing prevents overconfidence in the output. BCE Loss, as used in the original publication, employs a sigmoid layer in addition to cross-entropy loss for numerical stability.

The results indicate that Label Smoothing Focal Loss yields a validation mean F1 score of 0.7809, while BCE Loss yields a slightly higher F1 score of 0.7813. The difference is negligible (0.04 percentage points), suggesting that the two loss functions perform similarly for this task. However, BCE Loss results in significantly higher training and validation losses (0.1118 and 0.0981, respectively) compared to Focal Loss (0.0256 and 0.0279). This indicates that Focal Loss achieves better optimization stability, likely due to its ability to handle label imbalance, as described in the dataset's label imbalance in Section 2.2. While yielding a slightly lower F1 score, Focal Loss may be preferable when label imbalance is critical, as it reduces the risk of overfitting to dominant labels, such as label 1.

5.3 Learning Rate

Different learning rates were tested to assess their impact on model performance, with values of 0.0001, 0.001, 0.005, and 0.01. The learning rate controls the step size in gradient descent, and an optimal value balances convergence speed and stability.

A learning rate of 0.001 results in a validation mean F1 score of 0.6143, with training and validation losses of 0.0392 and 0.0391, respectively. In contrast, a learning rate of 0.0001 yields a lower F1 score of 0.5531 and higher losses (0.0567 and 0.0566), likely due to slow convergence from overly small updates. A learning rate of 0.005 achieves an F1 score of 0.5761, with losses of 0.0445 and 0.0436, indicating good but suboptimal performance compared to 0.001. The highest learning rate of 0.01 produces the same F1 score as 0.0001 (0.5531), with losses of 0.0567 and 0.0548, suggesting instability or overshooting during optimization.

These results suggest that a learning rate of 0.001 is optimal, striking a balance between stability and convergence speed. Learning rates higher or lower than this yield suboptimal performance, either slowing training or causing the model to diverge from the optimal solution.

Table 2: Ablation Study Results for MultiLabelClassifier Configurations

Configuration	**Train Loss**	**Val Loss**	**F1 Score**
Without Encoding	0.0174	0.0321	0.8054
With 2-Letter Encoding	0.0211	0.0213	0.8422
Focal Loss (label_smoothing=0.1)	0.0256	0.0279	0.7809
BCE Loss	0.1118	0.0981	0.7813
Learning Rate = 0.001	0.0392	0.0391	0.6143
Learning Rate = 0.0001	0.0567	0.0566	0.5531
Learning Rate = 0.005	0.0445	0.0436	0.5761
Learning Rate = 0.01	0.0567	0.0548	0.5531

5.4 Justification of the Best Model

Based on the experimental results, EfficientNet-B4 was selected as the best model for this study. Here are a few reasons why EfficientNet-B4 was chosen as the best model:

5.4.1 Superior Performance Efficient:

Net-B4 has superior performance compared to other models in terms of training loss, validation loss, and F1 score on test set (**0.908**). The F1 score of EfficientNet-B4 is also significantly higher, and the performance of the model continues to improve during the training of each epoch, indicating its strong generalization ability.

5.4.2 Advantage of composite expansion:

EfficientNet-B4 is compositely expanded in three aspects: depth, width and resolution. Traditional neural networks are usually extended in only one aspect, such as only increasing the number of layers of the network, which may lead to an increase in computation and deterioration in efficiency. EfficientNet-B4, through compound scaling, not only improves efficiency but also better extracts features from different scales of information, thereby enhancing classification accuracy.

5.4.3 Training stability and convergence speed:

Compared to other models, EfficientNet-B4 has a fast convergence speed. During the training process, both the training loss and the validation loss decrease rapidly, and the loss values always remain low. Unlike deep network models such as VGG16, EfficientNet-B4 avoids the problem of gradient vanishing because the network is too deep, which makes the training process more stable.

6 Conclusion

In this study, we explored the performance of multiple deep learning models in a multi-classification task, and comprehensively evaluated the training effectiveness, validation performance, and final prediction accuracy of each model on a specific dataset by comparing different deep neural network architectures. How to optimize the model structure to improve the effectiveness and stability of the multi-classification task is explored. Through comparative analysis, EfficientNet-B4 performs very well in the multi-label classification task.EfficientNet-B4 performs well in the F1 score, and is able to accurately recognize different labels, especially when the labels are unbalanced.EfficientNet-B4 uses the compound scaling approach to coordinate the expansion in depth, width and resolution, which enables the network to improve its performance in multiple dimensions, enabling more comprehensive extraction of image features and improved classification accuracy.

References

[1] Adane Nega Tarekegn, Mario Giacobini, and Krzysztof Michalak. A review of methods for imbalanced multi-label classification. *Pattern Recognition*, 118:107965, 2021.

[2] Jasmin Bogatinovski, Ljupčo Todorovski, Sašo Džeroski, and Dragi Kocev. Comprehensive comparative study of multi-label classification methods. *Expert Systems with Applications*, 203:117215, 2022.

[3] Joshua Ayeni. Convolutional neural network (cnn): the architecture and applications. *Appl. J. Phys. Sci*, 4:42–50, 2022.

[4] Xianfeng Ou, Pengcheng Yan, Yiming Zhang, Bing Tu, Guoyun Zhang, Jianhui Wu, and Wujing Li. Moving object detection method via resnet-18 with encoder–decoder structure in complex scenes. *IEEE Access*, 7:108152–108160, 2019.

[5] Laith Alzubaidi, Jinglan Zhang, Amjad J Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, José Santamaría, Mohammed A Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. *Journal of big Data*, 8:1–74, 2021.

[6] Paulo Freitas de Araujo-Filho, Georges Kaddoum, Mohamed Naili, Emmanuel Thepie Fapi, and Zhongwen Zhu. Multi-objective gan-based adversarial attack technique for modulation classifiers. *IEEE Communications Letters*, 26(7):1583–1587, 2022.

[7] Pan Zhang, Ling Yang, and Daoliang Li. Efficientnet-b4-ranger: A novel method for greenhouse cucumber disease recognition under natural complex environment. *Computers and Electronics in Agriculture*, 176:105652, 2020.

7 Appendix

7.1 Code

The data and code can be accessed at the following URL:
Google Drive Folder

7.2 How to run the code

1. **Open Google Colab and create a new notebook.**
2. **Upload the provided Jupyter notebook (** `540349603_540349681_530698180-COMP5329_2025S1A2-Code.ipynb`):
   - Go to `File > Upload notebook` and select the `.ipynb` file, or copy-paste the five code cells into separate cells.
3. **Enable GPU for faster training**:
   - Navigate to `Runtime > Change runtime type > Select GPU` under `Hardware accelerator > Click Save`.
4. **Install required libraries**:
   (a) In a new code cell, run:
   !pip install torch torchvision timm torchmetrics pandas matplotlib tqdm pillow numpy scikit-learn
   (b) Wait for the installation to complete before proceeding.
5. **Run: Imports**:
   (a) Execute the first cell to import libraries (e.g., `torch`, `pandas`, `google.colab.files`).
   (b) Ensure no errors occur. If a library is missing, install it with `!pip install <library>`.
6. **Run: Utility Functions**:
   (a) Execute the second cell to define utility functions (e.g., `set_seed`, `get_device`) and set up caption encoding.
   (b) Confirm it runs without errors (no output expected).
7. **Run: Clean and Visualize Dataset**:
   (a) **Upload Kaggle API Key**:
      - Run the cell; a prompt will appear to upload `kaggle.json`.
      - Download `kaggle.json` from your Kaggle account:
        – Go to `Kaggle > Account > API > Create New API Token`.

– Save the file locally and upload it when prompted.

(b) **The cell will**:

- Copy `kaggle.json` to `/.kaggle/` and set permissions.
- Download the dataset and unzip it to `/content/dataset`.
- Process `train.csv` and `test.csv`, saving cleaned versions as `train_cleaned.csv` and `test_cleaned.csv` in `/content/`.
- Generate a label distribution plot (`label_distribution.png`).

8. **Run: Model Comparison**:

(a) This cell trains and compares multiple models (`SimpleCNN`, `DeepCNN`, `ResNet18`, `VGG16`, `EfficientNet-B4`).

(b) Outputs a comparison plot (`model_comparison.png`) with train/validation loss and F1 scores.

(c) **Note**: Skip this step if you only need the best model's predictions (time-consuming).

9. **Run: Final Model and Predictions**:

(a) **Purpose**: Trains the best model (`EfficientNetWithCaptions`, combining `EfficientNet-B4` with caption features) and generates predictions.

(b) **Execution**:

- Loads `train_cleaned.csv` and `test_cleaned.csv`.
- Trains the model and saves it to `models/efficientnet_b4_timm_final.pth`.
- Generates a training metrics plot (`training_metrics.png`).
- Uses `MultiLabelPredictor` to predict on the test set, saving results to `submission.csv` (format: `ImageID,Labels`).

(c) **Note**: This is the recommended model for submission due to its use of both image and caption features.

AI statement

In this assignment, we used ai in a few ways to increase work efficiency. First, we asked AI to provide advice about top performance CNN models we can use for image classification and caption encoder. Additionally, because codes can be long and have hard architectures. AI can help us explain the codes, debug and refining our functions. Last but not the least, we also use AI like ChatGPT to revise our report writing and generate a tidy Latex format and help with inserting images and tables.