ÇUKUROVA UNIVERSITY

ENGINEERING AND ARCHITECTURE FACULTY

DEPARTMENT OF COMPUTER ENGINEERING

GRADUATION THESIS

DEVELOPMENT OF A HIERARCHICAL OBJECT DETECTION MODEL

By

Cansu KÜNAR

Salih Enes METİN

Advisor

Assoc. Prof. Dr. Serkan KARTAL

June 2024

ADANA

ABSTRACT 5
1. INTRODUCTION 6
2. DATASET 7
2.1. Dataset Introduction 7
2.2. Dataset Content 7
2.3. Editing the Dataset 8
3. PACKAGES AND LIBRARIES 10
3.1. Pandas 10
3.2. JSON 10
3.3. OS (Operating System) 10
3.4. Glob 11
3.5. Argparse 11
3.6. Warnings 11
3.7. train_test_split 11
3.8. shutil.copyfile 12
3.9. MMDetection 12
3.9.1. mmengine.config.DictAction 12
3.9.2. mmengine.config.Config 13
3.9.3. mmengine.runner.Runner 13
3.9.4. mmdet.registry.RUNNERS 14
3.9.5. mmdet.utils.setup_cache_size_limit_of_dynamo 14
4. MODEL CONFIGURATION AND PARAMETERS 16
4.1. Faster R-CNN R50 FPN 16
4.1.1. Faster R-CNN 16
4.1.2. ResNet-50 (R50) 16
4.1.3. Feature Pyramid Network (FPN) 16
4.2. Model Settings 16

4.2.1. Backbone 17
4.2.2. Neck 17
4.2.3. RPN Header 17
4.2.4. RoI Head 17
4.2.5. Training Settings 17
4.2.6. Test Settings 18
4.3. Dataset Settings 18
4.3.1. File Client Settings 18
4.3.2. Training Pipeline 18
4.3.3. Testing Pipeline 19
4.3.4. Train Data Loader 19
4.3.5. Validation and Testing Data Loader 19
4.3.6. Test Data Interference 19
4.4. Train Program 19
4.4.1. Learning Rate 20
4.4.2. Optimizer 20
4.4.3. Automatic Learning Rate Scaling 20
4.5. Default Runtime Settings 20
4.5.1. Default Hooks 20
4.5.2. Visualization Backends and Visualizer 21
4.5.3. Environment Configuration 21
4.5.4. Log Processor 21
4.5.5. Other Settings 21
5. METHODOLOGY 22
5.1. Convert Csv to Coco Json 22
5.1.1. Loading and Filtering the CSV Data 22
5.1.2. Sampling the Dataset 22

5.1.3. Reclassification of Labels 22
5.1.4. Generation of Class IDs 22
5.1.5. Splitting the Dataset 22
5.1.6. Processing and Converting Data into COCO Format 23
5.1.6.1. Initialization of Data Structures 23
5.1.6.2. Annotation and Image Processing 23
5.1.6.3. Compiling the Final JSON Structure 24
5.1.6.4. Output Notification 24
5.1.7. Saving the Formatted Data 24
5.2. create_trainfile 25
5.2.1. Configuration Loading 25
5.2.2. Directory Setup and Logging 25
5.2.3. Dynamic Configuration Generation 25
5.2.4. Configuration Parameter Adjustments 26
5.2.5. Configuration Dumping and Logging 26
5.2.6. Processing for Multi-Class Configurations 26
5.2.7. Finally 26
5.3. train_models_from_configs 27
5.3.1. Configuration File Validation 27
5.3.2. Reading Configuration Paths 27
5.3.3. Model Training Setup and Execution 27
5.3.4. Finally 28
6. MODEL HIERARCHY 29
7. RESULT 30
8. REFERENCES 31

ABSTRACT

This study examines the use of hierarchical models for fish species detection and classification in the commercial fishing industry. Using the Fishnet Open Images Database, advanced computer vision algorithms have been developed that can accurately and quickly identify fish species under low-visibility conditions and other challenging factors. The hierarchical model approach aims to increase model performance by addressing the unbalanced class distribution in the dataset. The results obtained will make significant contributions to the protection and management of marine resources by enabling more accurate detection of fish species in commercial fishing.

1. INTRODUCTION

In recent years, the need for effective monitoring and categorization of marine life has increased significantly, especially in the commercial fishing industry. To meet this need, electronic monitoring (EM) systems are implemented on commercial fishing vessels. These systems capture large amounts of video data via onboard cameras, which are then analyzed to ensure transparency and sustainability in the seafood market.

This project uses the Fishnet Open Images Database, a comprehensive dataset obtained from EM systems. This database contains 406,463 bounding boxes in 86,029 images captured by 73 different cameras. This represents the largest and most diverse collection of publicly available fisheries EM images and covers 34 object classes. The dataset presents unique challenges, such as visual similarities between fish species, unbalanced class distributions, adverse weather conditions, and intense crew activities, making it an ideal reference point for the development of robust computer vision algorithms.

The main purpose of this study is to improve the detection and fine categorization of fish species by using a hierarchical model for fish detection. The hierarchical model approach aims to improve overall model performance by providing a more accurate and faster classification of fish species. Using the Fishnet dataset, it is aimed at developing hierarchical models that can work effectively under low-visibility conditions and other complex factors. The study also aims to increase the accuracy and reliability of the models by addressing the unbalanced class distribution within the dataset.

In conclusion, this thesis aims to develop and evaluate hierarchical models for fish detection using the Fishnet Open Images Database. The research findings are expected to promote sustainability and transparency in the seafood industry by supporting more effective monitoring of commercial fishing activities.

2. DATASET

2.1. Dataset Introduction

The Fishnet Open Images Database [1] was utilized for this project. This dataset contains data obtained from electronic monitoring (EM) systems used on commercial fishing vessels for fish detection and fine categorization purposes. EM systems monitor fishing activities through cameras onboard ships and produce large amounts of video data to ensure transparency in the seafood market. This data is recorded and analyzed for review by experts on land.[2]

The Fishnet Open Images Database contains 406,463 bounding boxes in 86,029 images from 73 different electronic tracking cameras. This dataset is the largest and most diverse dataset of publicly available fisheries EM images, containing 34 object classes and presenting characteristic challenges such as visual similarity between fish species, unbalanced class distributions, harsh weather conditions, and busy crew activities. The dataset serves as a challenging benchmark for the development of computer vision algorithms, especially in areas such as fine categorization and detection in low-visibility conditions.

Figure 1: Images captured by the electronic monitoring system on a commercial fishing vessel. Crew members process the fish and the areas within the red box show the fish identified for analysis.

2.2. Dataset Content

The Fishnet dataset is currently limited to images from long-line tuna vessels in the western and central Pacific Ocean. More than 85% of the images in this dataset are of four major tuna species: Albacore, Yellowfin, Skipjack, and Bigeye. These four species are visually very similar to each other and account for the great majority of fish descriptions in the dataset. Beyond these four species, the dataset includes 25 additional fish species. Each of these types has been described in finer detail and called the “L1” tag set.

However, broader categories have been defined using logical groupings based on physical morphology. This grouping, based on the FAO ASFIS List of Species for Fishery Statistics

Purposes [3], divides related species into 12 more coarse classes. These classes are called the “L2” tag set and cover broader categories. For example, in this grouping, species such as billfish, marlin, and sailfish are grouped together based on similar physical characteristics. There is also a class called "OTH" (Other) at the "L2" level. This class includes fish tagged as "Unknown" at the "L1" level and all fish species with fewer than 1000 tags, but sharks are not included in this class due to their conservation importance.

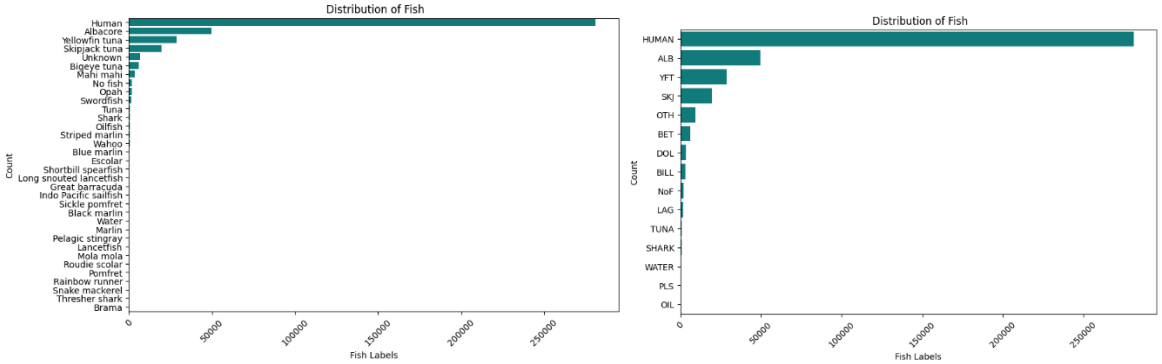

Figure 2:Initial state of Label 1 and Label 2 columns. The data includes various fish species as well as tags such as "Human", "No fish", "Unknown".

When the data set was examined, it was determined that there were the following columns:

img_id: Image ID is the unique identifier of each image.

bbox_id: The bounding box ID is the unique identifier of each bounding box.

x_min: The upper left corner x coordinate of the bounding box is the starting point of the box.

x_max: The bottom-right corner x coordinate of the bounding box is the end point of the box.

y_min: The upper left corner y coordinate of the bounding box is the starting point of the box.

y_max: The bottom-right corner y coordinate of the bounding box is the end point of the box.

label_l1: Fine class label refers to the detailed classification of fish species.

label_l2: Coarse class label refers to the classification of fish species into broader categories.

fao_mfa: FAO multi-frequency analysis code enables the classification of fish species according to standard codes defined in FAO's global fisheries database.

2.3. Editing the Dataset

Non-fish images were removed from the dataset used in this project. These images contain data tagged as “humans”, “unknown”, “no fish”, “water”, “oil” from data tag label_l1, and “TUNA” from data tag label_l2.

The reason for removing data labeled tuna is that the majority of images belong to tuna species, which make up more than 80% of the dataset. These species include albacore (ALB), bigeye (BET), yellowfin (YFT), and skipjack (SKJ). In our study, images labeled “TUNA” were not used in order to create models that distinguish different tuna species for which the species identity is known.

Organizing the data set in this way enables the model to produce more balanced and accurate results. This process reduces the unbalanced class distribution that the model will encounter during the training and testing processes and increases the performance of the model. In particular, it contributes to obtaining more effective results in areas such as fine categorization and detection in low-visibility conditions.

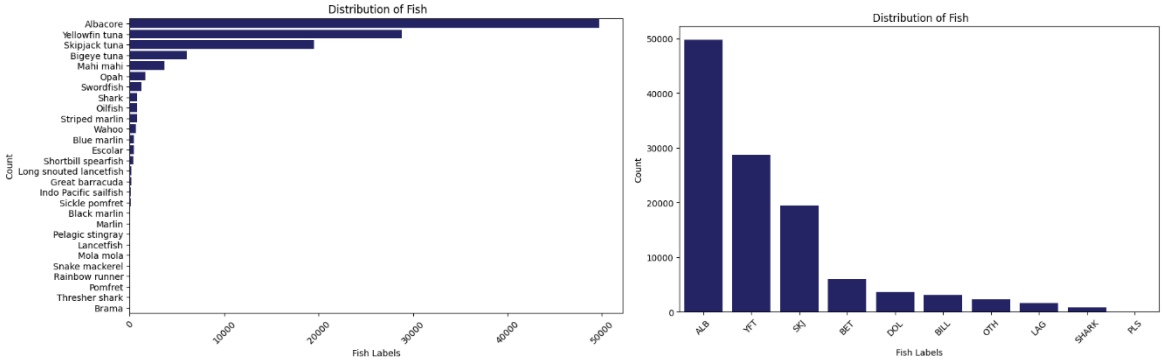

Figure 3:The state of Label 1 and Label 2 columns after the data set is edited. The dataset has been stripped of tags such as "Human", "No fish", "Unknown" to provide a more balanced and accurate representation of fish species.

3. PACKAGES AND LIBRARIES

3.1. Pandas

Pandas is a powerful library for data analysis and manipulation in Python. It facilitates and accelerates data processing processes by providing data structures and data analysis tools.

Pandas supports data processing operations such as dealing with missing and duplicate data, data filtering and sorting, data transformation and aggregation. Its key features include complete data processing, data acceleration and easy data transformation. Additionally, the flexible and extensible nature of Pandas makes it easy to retrieve data from various data sources and produce data output in various data formats. Pandas is widely used in data science and machine learning fields. It greatly speeds up and simplifies data analysis processes. [4]

3.2. JSON

JSON (JavaScript Object Notation) is an open, text-based and human-readable data exchange format derived from the JavaScript programming language. JSON provides data storage and communication in a lightweight format. Due to its syntax structure close to JavaScript, it is also supported by other programming languages other than JavaScript.

JSON is widely used for text-based data storage and communication. It works with an object structure containing name/value pairs, allowing data to be moved independently and lightweight.

JSON facilitates cross-platform data exchange and is widely used in web applications, APIs, and data storage. Thanks to these features, JSON simplifies and accelerates data communication and storage processes. [5]

3.3. OS (Operating System)

The OS module, which enables Python to interact with the operating system, performs tasks such as file and directory management, environment variables management and system searches. The OS module makes it easy to manage file paths using platform-independent commands and supports complex operations such as multi-process management.

The os.path submodule, part of the OS module, makes it easier to work with file and directory paths. This module is used for operations such as merging and parsing file and directory paths, obtaining absolute paths and checking file existence. Os.path supports platform-independent file path operations and ensures consistent operation across different operating systems.[6]

3.4. Glob

The Glob library is used to match file names according to certain patterns. Based on file naming patterns commonly used in the Unix shell, glob is useful for searching and listing files on the file system. It provides platform independence when working with file paths and patterns, making it easier to write code that runs on different operating systems.

The Glob library is ideal for automating file operations and increasing efficiency in large projects. For example, it makes it easier to find all files with a specific file extension or filter by file names. Thanks to these features, it provides efficient and effective file management in complex file systems.[7]

3.5. Argparse

The Argparse library is a module for creating command line interfaces. This library makes it easier to receive and process user input by defining command line options, arguments, and subcommands. Argparse is used to define command line arguments, specify whether they are required, and determine the data types of the arguments.[8]

3.6. Warnings

The Warnings library is used to manage runtime warnings in Python programs. This library provides functionality to create, display, and filter alerts to inform developers about potential issues. The Warnings module helps ensure that the software runs more reliably and error-free, while providing the flexibility to control how warnings are delivered to the user.[9]

3.7. train_test_split

Train-test split is a function taken from the model selection module of the Scikit-learn library. This function separates the dataset into training and testing subsets. It divides the dataset into two parts: features (X) and target variable (y) and creates training and testing subsets in specified proportions. In this project, the training section is further divided into training and verification once again.

The train_test_split function is a standard workflow step for training and evaluating machine learning models. This function is useful to evaluate the generalization ability of the model by creating different training and testing subsets each time, using the random sampling method.
The function is quite simple to use and is often the starting point in machine learning projects.

While dividing the dataset into training and testing subsets in certain proportions, it preserves the order of the samples in the dataset and uses the entire dataset. In this way, it is possible to evaluate the performance of the model and test it on different data subsets. [10]

3.8. shutil.copyfile

Copyfile is a function included in Python's standard library shutil. This function is frequently used during file operations and provides an effective solution for simple file copying operations. The copyfile function copies the specified source file (src) to the target file path (dst) and overwrites the target file if it exists.[11]

3.9. MMDetection

MMDetection is a library used for object detection and segmentation tasks and includes various modern algorithms. This library allows users to easily develop, customize and evaluate models thanks to its modular design.

MMDetection supports many different object detection and segmentation models and makes training and evaluation of these models easier. The library includes many core functions such as data preparation, model configuration, training cycle management, and evaluation reports. Additionally, it works with a variety of data sets and models, accelerating research and development processes.

MMDetection is extensible, allowing users to add their own custom object detection models or methods. The library offers a variety of tools and auxiliary functions to optimize training processes. This allows users to work efficiently on large data sets and complex models.

MMDetection is a powerful tool for researchers, data scientists, and machine learning engineers and is widely used in object detection and segmentation projects. This library is constantly updated and expanded to improve performance and simplify the development process.[12]

3.9.1. mmengine.config.DictAction

DictAction is a class taken from the configuration module of the MMEngine library. This class is used to convert command line arguments into a dictionary structure. It allows complex configuration settings to be easily passed from the command line.

When used with the argparse module, DictAction converts key-value pairs passed via the command line into a Python dictionary. This helps in dynamically updating configuration files and providing flexible configuration management. It is used to easily change and manage different configuration settings, especially in large projects and machine learning experiments.

Use of this class makes it easier to process command line arguments and pass them dynamically into configuration files. This provides flexibility when switching between different versions of the same configuration file. It allows users to quickly change configuration settings from the command line. This feature provides a great advantage, especially in rapid prototyping and testing processes.[13]

3.9.2. mmengine.config.Config

The Config class, taken from the configuration module of the MMEngine library, is used to load, process and manage configuration files. This class is a useful tool for editing and accessing settings and parameters used in projects.

The Config class loads configuration files from files written in Python syntax. This allows users to create flexible and dynamic configuration files. Configuration files can contain various information such as model settings, training parameters, bus settings, etc.

After loading the configuration files, the Config class makes the information about these files easily accessible. It also offers the possibility to combine different configuration files or change certain configuration settings with command line arguments. This feature simplifies configuration management in projects and enables quick switching of different experiment settings.[14]

3.9.3. mmengine.runner.Runner

The Runner class, taken from the runtime module of the MMEngine library, is used to manage the training and evaluation processes of machine learning models. The Runner class streamlines the training cycle, allowing the model to be trained and tested efficiently. This class provides functions and methods that cover training and evaluation processes. These processes include data loading, model training, validation, and evaluation.

The Runner class makes it easy to manage hyperparameters and settings used during the training process. It manages training and evaluation cycles, enables editing of hyperparameters and configuration settings, and monitors and reports progress during the training process. It can work compatible with different model architectures and data sets.

The Runner class is used to standardize and manage training processes, especially in large and complex machine learning projects. In this way, it helps users carry out their training and evaluation processes more efficiently and regularly. The flexibility and editing possibilities it provides at every stage of the training process play an important role in optimizing model performance.[15]

3.9.4. mmdet.registry.RUNNERS

The mmdet.registry.RUNNERS component, taken from the registry module of the MMDetection library, provides a central registry of the various "Runner" classes used in the MMDetection framework. This component defines Runner classes used to manage different training and evaluation processes and provides easy access to these classes.

The RUNNERS component provides centralized registration and management of all Runner classes defined and used in the MMDetection framework. This makes it easy for users to choose and use a particular Runner class according to their needs. For example, users can import the appropriate Runner class from this registry and use it in projects that require special training cycles or evaluation methods.

RUNNERS is a key part of MMDetection's modular structure, making it easy for users to extend and customize the framework. In this way, it becomes possible to select and use Runner classes suitable for different projects and usage scenarios. This component increases the flexibility of MMDetection and allows users to more easily manage training and evaluation processes to suit their needs.[16]

3.9.5. mmdet.utils.setup_cache_size_limit_of_dynamo

Mmdet.utils.setup_cache_size_limit_of_dynamo is a function taken from the utilities module of the MMDetection library. This function is used to set the cache size limit for the dynamo system. Dynamo system is a frequently used mechanism in big data and model operations, and this function provides cache management to optimize performance during these

operations.

The setup_cache_size_limit_of_dynamo function helps operations be more efficient by keeping memory usage under control, especially when working on large data sets or complex models. Limiting the cache size prevents memory overflows and ensures more efficient use of system resources.

This function is an important tool to increase the flexibility and performance of the MMDetection framework. Users can use this function to optimize the dynamo system and improve performance in large-scale data operations. Thus, it is possible to carry out training and evaluation processes more quickly and efficiently. This function optimizes memory management when working with big data, improving the overall performance of the system and making resource usage more efficient.[17]

4. MODEL CONFIGURATION AND PARAMETERS

4.1. Faster R-CNN R50 FPN

Faster R-CNN R50 FPN is a deep learning model used for object detection. This model consists of the combination of Faster R-CNN (Faster Region-based Convolutional Neural Network), a member of the "Region-based Convolutional Neural Network (R-CNN)" family, and ResNet-50 (R50) and Feature Pyramid Network (FPN) components.

4.1.1. Faster R-CNN

Faster R-CNN is a widely used model in object detection tasks and is known for its ability to make fast and accurate detections. This model is built on top of the R-CNN and Fast R-CNN models. It enables rapid and efficient extraction of candidate object regions using the Region Proposal Network (RPN). RPN identifies potential object regions using feature maps and then uses these regions for classification and bounding box refinement.[18]

4.1.2. ResNet-50 (R50)

ResNet-50 is a type of deep convolutional neural networks and is widely used in deep learning models. ResNet (Residual Network) is a structure that ensures that the network has very deep layers and facilitates the training of the model with connections called "residual connections". ResNet-50 is a 50-layer ResNet model and generally offers high performance and accuracy.[19]

4.1.3. Feature Pyramid Network (FPN)

FPN is an architecture that provides better object detection by combining feature maps at different scales. Using the pyramid structure, FPN combines features from different resolutions, allowing the model to detect small and large objects simultaneously. FPN, when combined with models such as Faster R-CNN, improves performance, especially when it comes to detecting small objects.[20]

4.2. Model Settings

The model is defined as FasterRCNN type. The data processor is of type DetDataPreprocessor, and the mean values for normalization of the images are [123.675, 116.28, 103.53] and standard deviation values are [58.395, 57.12, 57.375]. Images will be converted from BGR to RGB, and their size will be rounded to multiples of 32.

4.2.1. Backbone

The backbone is of ResNet type, 50 layers deep and four stages. The exit indices of the stages are determined as (0, 1, 2, 3). The first stage is frozen, where normalization is performed using BatchNorm (BN) and gradients are calculated during this normalization process. Additionally, the model is configured in pytorch style and initialized pre-trained with the torchvision://resnet50 checkpoint.

4.2.2. Neck

The neck part is FPN type and produces five different outputs by converting inputs from four different channels (256, 512, 1024, 2048) into 256 output channels.

4.2.3. RPN Header

The RPN header is of type RPNHead and contains 256 entries and 256 feature channels. The anchor generator used here is of type AnchorGenerator, with scales [8], rates [0.5, 1.0, 2.0] and steps [4, 8, 16, 32, 64]. The bounding box encoder is of type DeltaXYWHBBoxCoder, with target means set to [.0, .0, .0, .0] and target standard deviations set to [1.0, 1.0, 1.0, 1.0]. The classification loss is of type CrossEntropyLoss, using sigmoid and weight set to 1.0. Box loss is of type L1Loss, and the weight is set to 1.0.

4.2.4. RoI Head

The RoI header is of type StandardRoIHead and is determined using a single RoI extractor (SingleRoIExtractor). This extractor is of type RoIAlign, with output size set to 7 and sampling rate to 0. The output channels of the extractor are designated as 256 and the feature map steps are [4, 8, 16, 32]. The box header is of type Shared2FCBBoxHead and contains 256 input channels and 1024 output channels. The RoI feature size was determined as 7 and the number of classes was determined as 80. The box encoder is of type DeltaXYWHBBoxCoder, with target averages set to [0., 0., 0., 0.] and target standard deviations to [0.1, 0.1, 0.2, 0.2]. The regression is not class agnostic. The classification loss is of type CrossEntropyLoss, sigmoid is not used and the weight is set to 1.0. Box loss is of type L1Loss, and the weight is set to 1.0.

4.2.5. Training Settings

Training settings consist of two main parts: RPN and RCNN. The RPN section uses an assigner of type MaxIoUAssigner, with a positive IoU threshold of 0.7, a negative IoU

threshold of 0.3, and a minimum positive IoU of 0.3. The random sampler (RandomSampler) operates with 256 samples and a 50% positive rate. The limit is the allowable limit and the weight is non-negative. RPN recommendations are set at 2000 pre-NMS and a maximum of 1000 per image. The NMS type is set to nms with an IoU threshold of 0.5. The RCNN part uses an assigner of type MaxIoUAssigner and the positive IoU threshold is set as 0.5, the negative IoU threshold is 0.5, and the minimum positive IoU is 0.5. The random sampler (RandomSampler) operates with 512 samples and a 25% positive rate. The weight is non-negative, and errors are closed.

4.2.6. Test Settings

Test settings consist of two main parts: RPN and RCNN. The RPN portion is set to 1000 pre-NMS and a maximum of 1000 per image. The NMS type is set to nms with an IoU threshold of 0.5. The RCNN part is set to nms with a score threshold of 0.05, the NMS type is set to nms with an IoU threshold of 0.5, and a maximum of 100 results are retrieved per image. Additionally, soft NMS (soft_nms) is also supported.

4.3. Dataset Settings

The data set type used in the study was determined as CocoDataset. The data is stored in the data/fishnet/ directory. The mean values used for image normalization were defined as [123.675, 116.28, 103.53], and the standard deviation values were defined as [58.395, 57.12, 57.375]. Images will be processed in RGB format.

4.3.1. File Client Settings

For file client settings, the backend_args variable is set to None. This variable can be used to load the dataset from different storage solutions but is not used here.

4.3.2. Training Pipeline

The training pipeline consists of a series of operations to load and process data. As the first step, the LoadImageFromFile operation loads the image from the file. Following this, the LoadAnnotations operation loads the annotations and indicates the presence of bounding boxes. Next, the Resize operation scales the images to 800x800 and preserves the proportions. The PhotoMetricDistortion process applies photo-metric distortions to the image. Then, the RandomFlip operation randomly flips the image horizontally, which happens with a 50% probability. Normalize normalizes the image to the specified mean and standard deviation

values. The pad operation fills the image by rounding it to multiples of 32. Finally, the PackDetInputs process packages the processed images.

4.3.3. Testing Pipeline

The testing pipeline also consists of similar steps, but there are some differences. The LoadImageFromFile operation loads an image from the file. The resize operation scales images to 800x800 and preserves the proportions. Normalize normalizes the image according to the specified mean and standard deviation values. The PackDetInputs operation packages the processed images and adds meta information.

4.3.4. Train Data Loader

The training data loader is defined with certain parameters. batch_size is set to 2, num_workers is set to 2, persistent_workers is set to True. The sampler is of type DefaultSampler and selects randomly. batch_sampler is of type AspectRatioBatchSampler. dataset defines the training dataset and contains the necessary file paths.

4.3.5. Validation and Testing Data Loader

The validation data loader and the test data loader are defined similarly. However, the shuffle parameter is set to False, and the dataset runs in validation mode. CocoMetric evaluation metric is used for validation and testing. This metric evaluates bounding boxes and uses the specified annotation file.

4.3.6. Test Data Interference

While inference is performed on the test data set, the results are formatted and prepared for submission. The test data loader and evaluator are configured similarly to the verification data loader and evaluator. However, the results are only intended to be output in a specific format.

4.4. Train Program

The training program is defined with a configuration of type EpochBasedTrainLoop. This configuration specifies training for a maximum of 24 epochs. Verification will be carried out at the end of each period. The verification configuration is defined as type ValLoop, and the test configuration is defined as type TestLoop.

4.4.1. Learning Rate

A two-stage plan is implemented for the learning rate. In the first stage, using a LinearLR type learning rate determinant, the learning rate is initially increased linearly by a factor of 0.001. This increase continues from the 0th period to the 500th iteration. In the second stage, using a MultiStepLR type learning rate determinant, the learning rate is reduced by a factor of 0.1 at certain periods throughout the training process (in the 4th and 10th periods). This phase lasts from epoch 0 to epoch 24, and the learning rate reduction is performed on an epoch basis.

4.4.2. Optimizer

The optimizer is defined with a configuration of type OptimWrapper. This configuration uses the Stochastic Gradient Descent (SGD) algorithm. Learning rate was set to 0.0001, momentum to 0.9 and weight decay to 0.0001. These parameters determine how the model will be optimized during the training process.

4.4.3. Automatic Learning Rate Scaling

Default settings are defined for automatic learning rate scaling. The enable parameter determines whether autoscaling is enabled and is set to False here. The base_batch_size parameter determines the base mini-batch size, where it is set to 16 total with 8 GPUs, 2 samples per GPU.

4.5. Default Runtime Settings

This configuration determines the default runtime settings for MMDetection. The work scope is set to mmdet.

4.5.1. Default Hooks

Various transaction hooks have been defined. The early stopping hook (EarlyStoppingHook) follows the coco/bbox_mAP metric and has the patience value set to 6 and the minimum delta value to 0.005. The best model checkpoint (CheckpointHook) is recorded based on the coco/bbox_mAP metric, with larger values being better. The timer hook (IterTimerHook) times each iteration. The logger hook (LoggerHook) logs every 500 iterations. The parameter scheduler hook (ParamSchedulerHook) allows parameters to be updated. The sampler seed hook (DistSamplerSeedHook) sets the seed for the distributed sampler. The visualization hook (DetVisualizationHook) activates the visualization and draws it.

4.5.2. Visualization Backends and Visualizer

The visualization backends are designated as local visualization backend (LocalVisBackend) and Tensorboard visualization backend (TensorboardVisBackend). The visualizer (DetLocalVisualizer) runs using these backends and its name is set to visualizer.

4.5.3. Environment Configuration

The environment configuration was determined by disabling the cudnn_benchmark feature, setting the fork initiation method with multithreading configuration (mp_cfg), and setting the opencv_num_threads value to 0. Distributed configuration (dist_cfg) uses the NCCL backend.

4.5.4. Log Processor

The log processor (LogProcessor) is configured to process logging with a window size of 50 and by_epoch.

4.5.5. Other Settings

The daily level is determined as INFO. There is no loading (load_from) from a pre-trained model. There is no resumption of the training or testing process.

5. METHODOLOGY

5.1. Convert Csv to Coco Json

5.1.1. Loading and Filtering the CSV Data

The function begins by loading a CSV file from the specified `csv_path`. This CSV file contains the annotations for the dataset, including labels and bounding box coordinates. The function utilizes the `pandas` library for this purpose, which provides efficient data manipulation capabilities.[21] Once loaded, the data is filtered to include only those entries that belong to the specified `class_group`. This filtering is essential to focus the dataset on relevant classes for a specific training or validation task.

5.1.2. Sampling the Dataset

If a `sample_ratio` is provided, the function further reduces the dataset by randomly selecting a fraction of the data specified by the `sample_ratio`. This step is crucial for experiments where handling the full dataset is computationally expensive or unnecessary. The sampling is performed using a random seed (set to 42) to ensure reproducibility of the results.

5.1.3. Reclassification of Labels

When the `reclass` flag is set to True and a `mapping` dictionary is provided, the function will reclassify the labels according to the mapping specified. This is performed by mapping each label to a new category, with the ability to retain the original label if it does not exist in the mapping dictionary. This step is crucial for scenarios where labels need to be grouped or simplified, facilitating model training on a more generalized or specific task.

5.1.4. Generation of Class IDs

If no external `class_ids` are provided, the function generates these IDs autonomously by enumerating the unique labels after filtering and sampling. This ensures that each class has a unique identifier, which is critical for the correct annotation in the COCO format. This step is particularly important for maintaining consistency across different datasets and models.

5.1.5. Splitting the Dataset

The dataset is then split into training, validation, and test sets based on the `split_ratio`. This separation is critical for evaluating the model’s performance on unseen data, ensuring that the model is not overfitting to the training data. The split is performed using a stratified approach, if possible, to maintain an equal distribution of classes across the datasets.

5.1.6. Processing and Converting Data into COCO Format

5.1.6.1. Initialization of Data Structures

The function starts by initializing three main lists: `images`, `annotations`, and `categories`. These lists will hold the data structured according to the COCO format:[22]

- `images` contains dictionaries with details about each image, such as its ID, dimensions, and file name.
- `annotations` includes information about each object instance in an image, such as bounding box coordinates, category ID, and other details necessary for training object detection models.
- `categories` is a list of all the unique object categories in the dataset, each represented by a dictionary containing the category's ID and name.

An additional set, `image_ids_seen`, is used to track which images have already been processed to avoid duplicating image entries in the `images` list.

Before processing the actual annotations, the function first constructs the `categories` list. This is done by iterating over the `class_ids` dictionary, which maps class names to unique numerical IDs. For each class, a dictionary is created and appended to the `categories` list, ensuring each category is correctly represented in the final JSON.

5.1.6.2. Annotation and Image Processing

The function then iterates through each row of the dataset (provided as `data`), performing the following tasks:

1. Image Entry Creation: For each unique image ID (not previously seen in the `image_ids_seen` set), a dictionary is created with the image's ID, dimensions, and filename. This dictionary is then added to the `images` list and the ID is marked as seen.

2. Annotation Creation: For each object instance (row in the dataset), an annotation dictionary is created. This includes:
   - The ID of the annotation, which is sequentially assigned and incremented for each new entry.
   - The image ID to link the annotation to the correct image.
   - The category ID corresponding to the object's class, looked up from the `class_ids` dictionary.

- Bounding box coordinates (`bbox`), computed from the minimum and maximum x and y coordinates provided in the dataset. These are used to define the position and size of the object within the image.
- The area of the bounding box, calculated from the bounding box dimensions, which is crucial for some COCO metrics.
- A placeholder for segmentation data, which is set as an empty list since segmentation is not handled in this function.
- An `iscrowd` flag, set to 0 (assuming individual annotations and not crowd annotations).

5.1.6.3. Compiling the Final JSON Structure

After all images and annotations are processed, the function compiles them into a final dictionary that mirrors the structure expected by COCO. This includes keys for `images`, `annotations`, and `categories`, each linking to the respective lists filled during the function's execution.

5.1.6.4. Output Notification

Finally, the function outputs a confirmation message indicating that the JSON file has been created and specifies its location. This provides a clear indication to the user that the process has been completed successfully.

The `process_data` function is a crucial component for transforming annotated data into a structured format that facilitates efficient training and evaluation of object detection models, adhering to the standards set by the COCO dataset.

5.1.7. Saving the Formatted Data

Finally, the processed data for each set is saved into JSON files at the specified paths (`train_json_path`, `val_json_path`, `test_json_path`). The JSON format is widely used in machine learning tasks as it is straightforward to use and integrate with various tools and libraries.[23] Saving the data in this format allows for easy loading during the training and validation phases of model development.

This function encapsulates the entire process of preparing annotation data for object detection tasks, converting it into a standard format, and ensuring that the data is ready for use in training robust models.

5.2. create_trainfile

The `create_trainfile` function orchestrates the configuration of multiple model training scenarios based on a hierarchical model structure and generates corresponding COCO JSON files for each configuration. This function is essential for managing multiple training setups systematically, ensuring each scenario is prepared with its specific data and settings.

5.2.1. Configuration Loading

Initially, the function reads a configuration file (`cfg_path`) to load all necessary parameters for the setup, using `Config.fromfile()`. This configuration file contains paths, image dimensions, hierarchy information, and other metadata required for processing. The function handles the initial setup by extracting these parameters directly from the configuration object (`cfg`), setting the stage for subsequent operations.

5.2.2. Directory Setup and Logging

A directory specified by `base_dir` is confirmed to exist or created using `os.makedirs()`, ensuring there is a location to store the generated files and logs. A log file (`config_paths_log`) is then prepared to record the paths of all generated configuration files. This is crucial for tracking and managing configurations systematically across multiple training scenarios.

5.2.3. Dynamic Configuration Generation

The core of the function lies in dynamically generating training, validation, and testing configurations based on a predefined hierarchy. This is achieved through nested loops iterating over the depths and lengths of the hierarchy as specified in `cfg.hierarchy`. For each configuration:

1. **Directory and Path Setup**: Specific directories for each configuration are created to store the corresponding JSON files and model results. This organization is essential for managing multiple models and their outputs efficiently.

2. **COCO JSON File Creation:** The `sample_and_convert_csv_to_coco_json` function is invoked to create COCO-formatted JSON files for training, validation, and testing datasets. This process involves specifying paths for each JSON file, setting the appropriate class groups based on the current hierarchy, and other parameters like image dimensions and class mappings.

5.2.4. Configuration Parameter Adjustments

After generating the JSON files, the function adjusts several parameters within the configuration object (`cfg`):

- Model Class Count: The number of classes (`num_classes`) is updated based on the length of the `classes` dictionary returned from the JSON file creation function. This adjustment is crucial for configuring the model correctly according to the number of target classes in the current scenario.
- Metadata and Data Loader Settings: The `metainfo` attribute, which contains class names, is set for different components of the data loader configuration (`train_dataloader`, `val_dataloader`, and `test_dataloader`). This ensures that each data loader component is aware of the classes involved in the current training scenario.
- Annotation File Linking: Paths to the generated COCO JSON files are set for the training, validation, and testing data loaders and evaluators. This linking is essential for ensuring that each model training session utilizes the correct dataset.

5.2.5. Configuration Dumping and Logging

Each dynamically adjusted configuration is then saved back to a Python file (`config.py`) in its respective directory. This file serves as the standalone configuration for training the model under the current setup. The path to this configuration file is also logged in the `config_paths_log`, allowing for easy access and management of different model configurations.

5.2.6. Processing for Multi-Class Configurations

In addition to the primary hierarchical setups, the function also handles multi-class configurations specified in the hierarchy. This involves similar steps of directory creation, JSON file generation, configuration adjustments, and logging, tailored for scenarios where multiple classes are managed collectively rather than in a hierarchical division.

5.2.7. Finally

The `create_trainfile` function is a comprehensive tool for setting up and managing diverse training scenarios based on a hierarchical or multi-class structure. By automating the generation of configurations and linking them with the appropriate data sets, this function

significantly simplifies the process of training multiple models, making it highly scalable and efficient for large-scale projects.

5.3. train_models_from_configs

The `train_models_from_configs` function is designed to automate the training of detection models from multiple configuration files. This function streamlines the process of reading configurations, setting up the training environment, and executing the training process. Below is a detailed explanation of each step in the function, including how parameters are utilized:

5.3.1. Configuration File Validation

The function starts by loading the master configuration file using `Config.fromfile(cfg_path)`, which contains all necessary meta-information for the training process, including paths to individual model configuration files (`config_paths_file`). This initial step is crucial as it establishes the base settings and paths that guide the subsequent operations.

5.3.2. Reading Configuration Paths

Upon confirming the existence of the `config_paths_file`, the function opens and reads the file, extracting a list of paths to the individual model configuration files. These paths are crucial for accessing and loading each model's specific settings and parameters, which include model architecture, training parameters, and dataset paths.

5.3.3. Model Training Setup and Execution

For each configuration path extracted from the `config_paths_file`:

1. Configuration Existence Check: The function verifies the existence of each configuration file. If a file is missing, it outputs a notification and skips to the next configuration. This step prevents the training process from attempting to load a non-existent configuration, which would result in an error.

2. Loading Model Configuration: Once a configuration file's existence is confirmed, it is loaded using `Config.fromfile(config_path)`. This operation populates a configuration object with all settings specified in the file, including data loaders, model architecture, optimization parameters, and training schedule.[24]

3. Model Training Initialization: A training runner (`Runner`) is initialized with the

loaded configuration. The `Runner.from_cfg(config)` method sets up the training environment according to the configuration, preparing the model, datasets, optimizer, and learning rate scheduler as defined.

4. Training Execution: The training process is initiated using `runner.train()`. This method engages the training loop, which iteratively trains the model over the dataset according to the epochs and batches specified in the configuration. During training, model performance metrics are calculated, and model weights are updated iteratively.

5. Completion Notification: Upon the completion of the training for a given model, a message is printed to indicate that training has finished. This message includes the path of the configuration file, providing a clear indication of which model configuration has been processed.

5.3.4. Finally

The `train_models_from_configs` function effectively automates the task of training multiple models by sequentially loading and executing each model's specific configuration. This automation is particularly useful in environments where numerous models need to be trained, such as in research settings or in development pipelines where experiments with different hyperparameters and architectures are common. By centralizing the control over the training process through a single master configuration and multiple model-specific configurations, this function enhances the scalability and manageability of model training operations.

6. MODEL HIERARCHY

A "hierarchy" parameter has been added to the configuration file to define the model structure. This parameter starts with the "depth" variable, which specifies the overall depth of the model hierarchy. For each depth, layers containing a certain number of models are defined. These layers are named "layer1", "layer2", etc., and the number of models in each layer is specified with the "length" parameter. Each model is named sequentially, such as "depth1_bmodel_1", "depth1_bmodel_2", etc., and these models can be assigned to specific classes or classes can be renamed. Finally, models that cover more than one class are named "multiclassm1", "multiclassm2", etc., and these models group certain classes. This configuration enables the model to effectively manage data relationships of varying depth and complexity.

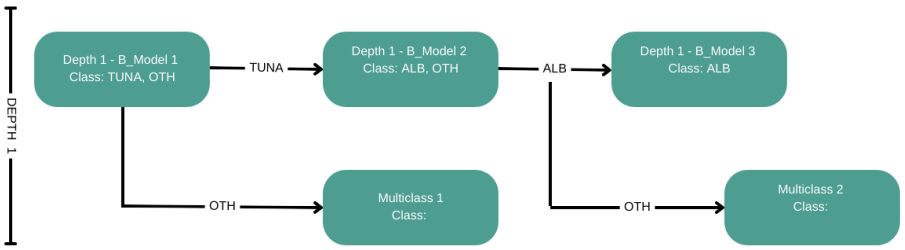

Figure 4:Hierarchical model structure.

7. RESULT

In this hierarchical model study, the testing process will commence with depth1_bmodel_1. If the result is 'true', the process will proceed to depth1_bmodel_2; if 'other', it will divert to the multiclassm1 model. Upon reaching depth1_bmodel_2, if the result is 'alb', it will advance to depth1_bmodel_3; if 'other', specifically '1', it will shift to the multiclassm2 model. Thus, each image is validated by a minimum of two models. Structuring the model in this layered manner enhances its performance on challenging hierarchical datasets.

The objective of this study is not solely to achieve high accuracy rates but to compare the hierarchical model against conventional models in two distinct scenarios: one involving a standard model performing multiple detections, and the other our hierarchical model. Both models were trained using the same parameters and datasets. Table 1 displays the bbox map results for five models within the hierarchical method, illustrating a decrease in performance as data density reduces, with the best results obtained at the highest layer. Table 2 compares the success rates between the hierarchical method and the traditional multiple detection model. It is evident that the hierarchical approach is more successful.

This method facilitates more effective learning of dominant classes within datasets while achieving optimal performance for less prevalent classes.

Model	Bbox Map
depth1bmodel1	0.258
depth1bmodel2	0.219
depth1bmodel3	0.196
multiclassm1	0.066
multiclassm2	0.179

Figure 5:Bbox Map performances of different depth and multi-class models of the hierarchical model. While Depth1-B_Model1 shows the highest success, multiclass-1 shows the lowest success.

Model	Bbox Map
FasterRCNN	0.102
Our Hierarchical Method	0.220

Figure 6: Bbox Map comparison of FasterRCNN and Hierarchical Model. The hierarchical model shows a higher success rate than FasterRCNN.

8. REFERENCES

[1] The Nature Conservancy (2021): Fishnet Open Images Dataset <v0.2.0> The Nature Conservancy. Dataset. https://www.fishnet.ai/

[2] Mark Michelin, Matthew Elliott, Max Bucher, Mark Zimring, and Mike Sweeney. (2018). Catalyzing the growth of electronic monitoring in fisheries.

[3] Food and Agriculture Organization of the United Nations Fisheries and Aquaculture Department. (2020). Fishery fact sheets collections: Asfis list of species for fishery statistics purposes.

[4] McKinney, W., & Team, P. D. (2015). Pandas-Powerful python data analysis toolkit. Pandas—Powerful Python Data Analysis Toolkit, 1625.

[5] Lennon, J. (2009). Introduction to JSON. In Beginning couchdb (pp. 87-105). Berkeley, CA: Apress.

[6] os — Miscellaneous operating system interfaces. Python documentation. https://docs.python.org/3/library/os.html

[7] glob — Unix style pathname pattern expansion. Python documentation. https://docs.python.org/3/library/glob.html

[8] argparse — Parser for command-line options, arguments and sub-commands. Python documentation. https://docs.python.org/3/library/argparse.html#module-argparse

[9] warnings — Warning control. Python documentation. https://docs.python.org/3/library/warnings.html

[10] train_test_split. scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

[11] shutil — High-level file operations. Python documentation. https://docs.python.org/3/library/shutil.html

[12] MMDetection — MMDetection 3.3.0 documentation. https://mmdetection.readthedocs.io/en/dev-3.x/overview.html

[13] Config — mmengine 0.10.4 documentation. https://mmengine.readthedocs.io/en/latest/advanced_tutorials/config.html

[14] Config — mmengine 0.10.4 documentation. https://mmengine.readthedocs.io/en/latest/advanced_tutorials/config.html

[15] Runner — mmengine 0.10.4 documentation. (https://mmengine.readthedocs.io/en/latest/tutorials/runner.html

[16] Registry — mmcv 1.7.1 documentation. https://mmcv.readthedocs.io/en/master/understand_mmcv/registry.html

[17] mmdet.apis — MMDetection 3.0.0 documentation. https://mmdetection.readthedocs.io/en/3.x/api.html#module-mmdet.utils

[18] Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. Advances in neural information processing systems, 28.

[19] Koonce, B., & Koonce, B. (2021). ResNet 50. Convolutional neural networks with swift for tensorflow: image recognition and dataset categorization, 63-72.

[20] Lin, T. Y., Dollár, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2017). Feature pyramid networks for object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2117-2125).

[21] User Guide — pandas 2.2.2 documentation. https://pandas.pydata.org/docs/user_guide/

[22] COCO format - Rekognition. https://docs.aws.amazon.com/rekognition/latest/customlabels-dg/md-coco-overview.html

[23] JSON explained. (n.d.). ai-jobs.net. https://ai-jobs.net/insights/json-explained/

[24] Config — mmcv 1.7.1 documentation. https://mmcv.readthedocs.io/en/master/understand_mmcv/config.html