Watermark Detection in Images

This report outlines the key components and decisions involved in developing the Watermark Detection in Images project, a FastAPI-based service that uses a Faster R-CNN model to detect watermarks in images. The project was developed as part of the 99 Group Role Challenge Test for the Senior Machine Learning Engineer position. Below are the main stages of the project, from data labeling to deployment, along with challenges encountered during the process. Additionally, potential future improvements are discussed to further enhance model performance and scalability.

Data Preprocessing

Data Labeling

To train the watermark detection model, a dataset containing both watermarked and non-watermarked images was labeled. Label Studio, an open-source tool for data labeling, was used to define bounding boxes around the visible watermarks in each image. This step ensures that the model has a clear understanding of the regions containing watermarks.

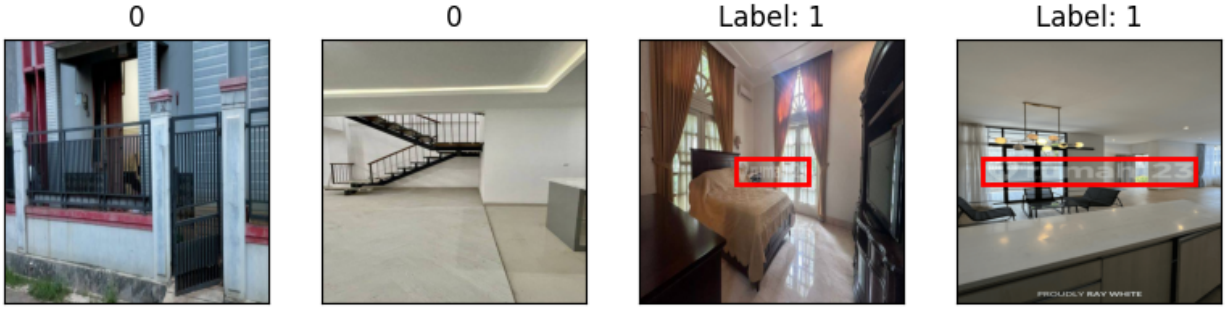

Image Resizing

The resize step needed because of the resize transform applied in fasterRCNN in the torchvision detection module. Images (and bounding box) were resized to (800, 800).

Dataset Splitting

The labeled dataset was split into 70% for training, 15% for validation and 15% for testing.

Model Selection

Faster R-CNN in PyTorch was chosen for this project due to its effectiveness in object detection tasks, especially when dealing with bounding boxes. It leverages region proposal networks to efficiently predict object locations and is widely used in scenarios requiring high accuracy in object localization. Given that the task involves detecting small watermarks in images, the model's robust architecture and pre-trained versions available in PyTorch make it suitable for this task.

However, it's important to note that there is potential for improvement. Exploring other architectures could be beneficial. These models might provide faster inference times or better performance, especially when detecting small, intricate details like watermarks. Future iterations of the project should evaluate multiple architectures to find the most optimal solution for this specific use case.

Performance Metrics

The following metrics were used to evaluate the model's performance:

Dataset	Avg. IoU	Precision	Recall	F1-Score
Train/Validation	0.4980	0.2381	0.5556	0.3333
Test	0.4954	0.3158	0.8571	0.4615

The performance was measured on all predicted bounding boxes. However, since there is only one watermark per image, it is recommended to calculate the performance metrics only for the **top 1 bounding box** based on the confidence score to provide a clearer view of how the model performs on the most confident predictions.

Also, the test performance is greater than the validation performance, it can indicate several things: the test set is easier or the test set is too small.

Deployment Steps

For detailed deployment instructions, please refer to the **README** file.

In brief, the project can be deployed:

- Locally using **FastAPI**
- Containerized using **Docker** and then run in any environment
- Deployed on a **Kubernetes** cluster (Minikube was used during development)

Challenges Faced

1. **Data Annotation**
   Labeling objects in images is time-consuming and prone to human error. Precise bounding boxes and class labels are needed for high accuracy, and inconsistent labeling can hurt model performance.
2. **High Computational Cost**
   We develop our model using kaggle GPU T4 x 2 since our local machine is not reliable.

Future Improvements

These improvements will enhance the project's scalability, robustness, and ease of deployment, making it more suitable for production environments and continuous development.

1. **Unit Testing**
   Add unit tests to ensure code reliability. This helps catch bugs early, especially when changes are made. Key parts to test include image preprocessing, model loading, and prediction outputs.
2. **Pre-commit Hooks**
   Use pre-commit hooks to enforce code quality before changes are committed. Tools like `black` or `flake8` can ensure code is well-formatted and secure, preventing potential issues.
3. **Load Testing**
   Perform load testing to check how the API handles heavy traffic or large images. This helps identify performance bottlenecks, such as slow response times or memory usage issues, ensuring the system can scale effectively.
4. **Explore Architecture Models**
   While Faster R-CNN was used, exploring other architectures like YOLO or U-Net could improve speed and accuracy, especially for smaller objects like watermarks. Testing different models might yield better results.