Improving Performance of Detecting Military Vehicles' through Super Resolution

San Kim
Mokmalla, Korea Univ.
Seoul, Republic of Korea
san2710@korea.ac.kr

Chanha Park
Mokmalla, Korea Univ.
Seoul, Republic of Korea
qkrcksgk02@korea.ac.kr

Yurim Lee
Mokmalla, Korea Univ.
Seoul, Republic of Korea
leeyou6757@gmail.com

Yejin Hong
Mokmalla, Korea Univ.
Seoul, Republic of Korea
lilyhong511@korea.ac.kr

Abstract

In modern warfare, accurately assessing the opponent's military capabilities is crucial for strategic planning. Traditional methods of analyzing satellite images for identifying and tracking military assets are often time-consuming and error-prone. This paper explores the use of advanced artificial intelligence (AI) and machine learning techniques to automate and enhance the accuracy of these processes. We compare three AI models—EDSR, SRGAN, and our custom CycleGAN—in their ability to generate high-resolution images from low-resolution satellite photographs. Our experiments demonstrate that while EDSR and SRGAN are effective, CycleGAN offers a superior balance of performance and efficiency. It achieves higher PSNR values with fewer parameters, making it a robust option for applications requiring high-quality image generation with limited computational resources.

Building upon these findings, we integrate the super-resolution (SR) images into object detection tasks to evaluate their impact on identifying military platforms using the xView dataset. The results reveal that SR techniques significantly improve the detection accuracy of military assets, with EDSR and CycleGAN particularly excelling in restoring critical image details that enhance object detection performance. By combining SR methods with state-of-the-art object detection algorithms, this study demonstrates the potential of AI to revolutionize military decision-making by providing timely, precise intelligence. The proposed approach not only enhances situational awareness but also offers a scalable solution for strategic advantage in resource-constrained environments.

ACM Reference Format:
San Kim, Chanha Park, Yurim Lee, and Yejin Hong. 2024. Improving Performance of Detecting Military Vehicles' through Super Resolution. In . ACM, New York, NY, USA, 17 pages.

1 Introduction

Object detection technology has evolved rapidly over the past decade, with applications in a variety of fields, including defense and military operations, as well as civilian applications. In a military environment, the ability to detect, classify, and track military vehicles is essential for military situational awareness and subsequent decision-making and mission execution. Based on the importance of object detection for military vehicles, this paper will discuss ways to improve it.

Modern military systems require robust and reliable systems for identifying and monitoring military assets such as armored vehicles, tanks, military trucks, missile launchers, military aircraft, military helicopters, warships, etc. An object detection system designed for military applications will need to perform well in a variety of environments, including low light and complex backgrounds. This paper emphasizes the importance of advancing the capabilities of systems with respect to the importance of object detection for military vehicles. Object detection for military vehicles serves several functions in the military, including surveillance and reconnaissance, identification and threat analysis of enemy military vehicles, targeting of attacks, border security and smuggling surveillance, and is of such importance that it can be said to be directly related to national security.

Object detection for military vehicles serves several functions in the military, including surveillance and reconnaissance, identification of enemy military vehicles and threat analysis, targeting of attacks, border security and smuggling surveillance, and is a critical task that can be said to be directly related to national security.

Super Resolution, or SR, technology is the conversion of low-resolution images to high resolution. Higher-resolution images have more detail and less noise than lower-resolution images, which improves visual clarity and quality. Since SR technology based on deep learning can reconstruct images based on understanding the surrounding context and achieve natural and vivid results, we envisioned that object detection on satellite images that have been subjected to SR would yield better results than object detection on low-resolution satellite images that have not been subjected to SR. In other words, SR would enable better detection of small military vehicles and complex environments. The importance of object detection in military vehicles for military purposes is obvious. Especially in environments where low-quality satellite imagery is the primary data source, the application of Super Resolution technology is expected to significantly improve detection accuracy and support military decision-making.

In order to maximize the object detection performance of military vehicles in a military environment, it is necessary to systematically compare and evaluate various SR models. SR models can be broadly

Class76 Capstone, December 2024, Korea University

categorized into Convolutional Neural Network (CNN)-based models that process static images and Generative Adversarial Netwroks (GAN)-based models that have recently gained attention. Representative CNN-based models include EDSR, and GAN-based models include SRGAN and CycleGAN. Based on previous studies, one of the objectives of this research is to select the optimal model considering the processing performance, learning efficiency, and ease of application of these models, and to devise a method that can be effectively applied to satellite images for military purposes.

In order to quantitatively evaluate the specific impact of SR on the performance of military satellite imagery-based object detection, this paper analyzes key performance metrics such as accuracy, precision, and recall of object detection on data before and after applying SR. The change in performance of object detection models such as YOLO before and after SR processing is evaluated to demonstrate the practical contribution of SR techniques. We optimize the performance of SR models by reflecting the characteristics of satellite imagery data and military requirements, and select appropriate datasets and training models for pretraining and fine-tuning. This improves the SR-trained detection model to be optimized on military datasets. This research focuses on solving the problem of low resolution of satellite imagery and improving the reliability and accuracy of military object detection.

2 Related Works

For super resolution and object detection in military satellite imagery, we analyzed five papers. These five studies collectively underscore the advancements in super-resolution and object detection techniques. The integration of super-resolution methods like SRGAN and VDSR with object detection frameworks such as YOLO and Faster R-CNN offers significant potential for enhancing the performance of military satellite imagery analysis. By leveraging high-resolution images and efficient detection models, these approaches address critical challenges in identifying and analyzing military assets in satellite data, paving the way for more accurate and reliable results.

Moreover, these papers are closely interlinked in their contributions to the fields of super resolution and object detection. Ledig et al.'s SRGAN provides a foundation for generating high-quality upscaled images, which is further evaluated in the work of Shermeyer and Van Etten, demonstrating the tangible benefits of super resolution in object detection performance. Concurrently, Zou et al.'s survey of object detection techniques highlights the historical progression and the emergence of deep learning frameworks like YOLO, introduced by Redmon et al., and Faster R-CNN, proposed by Ren et al., as the backbones of real-time and high-accuracy object detection. The combination of SRGAN-generated high-resolution images, YOLO's real-time detection capabilities, and Faster R-CNN's high-precision region proposals is particularly powerful for applications requiring the analysis of large-scale satellite imagery.

Our work, *Improving Performance of Detecting Military Vehicles through Super Resolution*, builds upon these studies by specifically targeting the detection of military vehicles in satellite imagery. We leverage the super-resolution capabilities demonstrated by SRGAN and VDSR to enhance image quality, while adopting state-of-the-art object detection frameworks like YOLO and Faster R-CNN to accurately identify and classify military vehicles. By bridging the advancements in super resolution and object detection, our research aims to address the unique challenges posed by military satellite imagery, such as the small size, occlusion, and complex backgrounds of military vehicles. This integration leads to improved detection performance, enabling more precise and reliable identification of military assets in high-stakes scenarios.

2.1 Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network

The paper *Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network* introduces SRGAN, a generative adversarial network for single-image super-resolution. SRGAN aims to enhance low-resolution images to high-resolution ones with photorealistic details, specifically for high upscaling factors (4×). Traditional super-resolution methods often optimize for mean squared error (MSE), resulting in high peak signal-to-noise ratio (PSNR), but fail to produce perceptually pleasing textures.

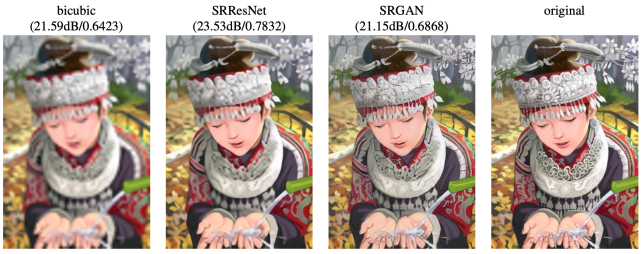

Figure 1: ŏ0d74 Super-resolution result of SRGAN compared with existing algorithms.

SRGAN addresses this limitation by employing a perceptual loss function that combines adversarial loss, which encourages the generator to produce images indistinguishable from real images, and content loss based on high-level features from the VGG network. This design allows SRGAN to generate sharper and more realistic textures compared to prior methods.

The architecture of SRGAN consists of two primary components: a generator and a discriminator. The generator, built on a deep residual network, focuses on reconstructing high-resolution images by capturing fine details and textures. Meanwhile, the discriminator evaluates the authenticity of the generated images by distinguishing them from real high-resolution images. This adversarial training process ensures that the generated images achieve both structural consistency and visual fidelity.

In addition to structural design, SRGAN leverages a perceptual loss metric, which includes content loss calculated from the high-level feature maps of a pre-trained VGG network. By aligning generated images with perceptual quality rather than pixel-level accuracy, SRGAN significantly improves the realism of its outputs. This is further validated through Mean Opinion Score (MOS) evaluations, where human participants consistently rated SRGAN-generated images as more photorealistic compared to other state-of-the-art methods.

One notable component of SRGAN's success is its ability to address the texture synthesis challenge in high upscaling scenarios. While traditional methods, such as bicubic interpolation, produce blurred textures, SRGAN enhances fine details, making it particularly useful for applications requiring high precision, such as satellite imagery analysis, medical imaging, and surveillance systems. Furthermore, the incorporation of SRResNet within the framework ensures high PSNR and SSIM scores, demonstrating a balance between perceptual quality and quantitative performance.

The advancements introduced by SRGAN extend beyond technical innovation, offering practical applications in diverse fields. In satellite imagery, SRGAN aids in the reconstruction of high-resolution images critical for detecting small objects or features, such as vehicles or infrastructure. In medical imaging, the enhanced resolution facilitates the accurate diagnosis of conditions by improving the clarity of critical image details. Surveillance systems benefit from SRGAN's ability to restore degraded video feeds, enabling better object identification and activity monitoring.

Moreover, SRGAN underscores the importance of moving beyond traditional pixel-based metrics like PSNR. By prioritizing perceptual quality, the framework highlights a shift in focus toward human-centric evaluations of image reconstruction tasks. This perspective is increasingly relevant as AI-driven applications prioritize user experience and real-world utility over theoretical performance metrics.

Despite its successes, SRGAN is not without limitations. The adversarial training process is computationally intensive, requiring significant resources for effective convergence. Additionally, the model's reliance on pre-trained networks, such as VGG, introduces dependencies that may not always align with specific domain requirements. Future iterations of SRGAN could explore lightweight architectures and domain-specific adaptations to overcome these challenges.

In conclusion, SRGAN represents a significant milestone in super-resolution research, combining technical ingenuity with practical utility. Its ability to generate photorealistic high-resolution images has broad implications, particularly in fields where image quality directly impacts decision-making processes. As research in this area evolves, SRGAN's foundational contributions will continue to inspire advancements in image synthesis and restoration.

2.2 The Effects of Super-Resolution on Object Detection Performance in Satellite Imagery

The paper, *The Effects of Super-Resolution on Object Detection Performance in Satellite Imagery*, presents an in-depth exploration of how super-resolution (SR) techniques enhance object detection capabilities in satellite imagery. Super-resolution, a process that reconstructs higher-resolution images from lower-resolution inputs, has emerged as a critical tool in remote sensing applications, particularly for identifying small or complex objects in imagery data.

This study employs two notable SR approaches, Very Deep Super-Resolution (VDSR) and a custom Random Forest Super-Resolution (RFSR) framework. These methods are applied to satellite images across five varying ground sample distances (GSDs), ranging from 30 cm to 4.8 meters. The research further integrates these SR-enhanced images with the SIMRDWN detection framework, a composite system that includes advanced object detection models like YOLO and SSD, to evaluate the impact of SR at enhancement scales of 2×, 4×, and 8×.

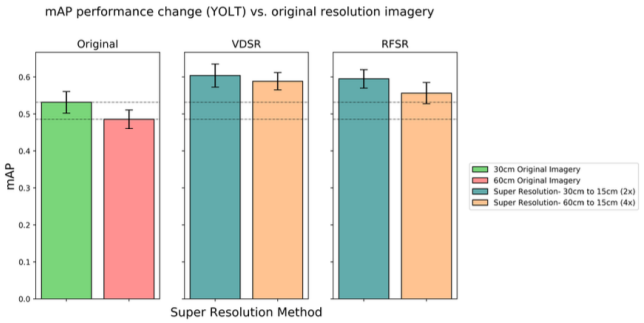

Figure 2: Performance boost of enhancing 30 and 60 cm imagery to 15 cm GSD.

At its core, SR addresses the fundamental challenge of resolution constraints in satellite imagery. High-resolution images are essential for detecting small, densely packed, or occluded objects. For instance, vehicles, equipment, or infrastructure components often appear as indistinct shapes in coarse-resolution images. By enhancing these images through SR, the study demonstrates substantial improvements in object detectability and classification accuracy.

The SR-enhanced images show marked improvements in mean average precision (mAP) across varying GSDs. Super-resolving 30 cm imagery to 15 cm resulted in mAP increases ranging from 13% to 36%, particularly for smaller objects such as vehicles. Similarly, 60 cm imagery upscaled to 15 cm yielded mAP improvements of up to 20%. These gains underscore SR's potential to extract critical details that are otherwise lost in native resolution imagery.

The integration of SR-enhanced images with advanced object detection frameworks highlights the compatibility and efficacy of these methods. YOLO consistently outperformed SSD in both accuracy and robustness, particularly when detecting smaller objects in cluttered environments. The grid-based architecture of YOLO allows for precise localization, while SR preprocessing amplifies its detection capabilities. This synergy between SR and YOLO provides a robust approach to addressing the challenges of remote sensing data.

The findings of this study extend beyond theoretical evaluation, offering practical implications for diverse fields. In urban monitoring, SR-enhanced satellite imagery enables precise mapping of infrastructure, supporting efficient urban planning. For disaster response, higher-resolution imagery improves the detection of damaged areas and critical assets, facilitating targeted relief efforts. Moreover, in military applications, the ability to identify and classify vehicles, equipment, and personnel in complex terrains is significantly enhanced, ensuring more effective decision-making.

Despite its advantages, SR is not without limitations. The computational intensity of SR algorithms, especially for high upscaling factors, presents challenges for real-time applications. Additionally,

the benefits of SR diminish at coarser resolutions (e.g., 4.8 meters), where the lack of high-frequency details limits the reconstruction process. This study suggests that future research should focus on optimizing SR algorithms to balance computational efficiency and performance, potentially through hybrid models or domain-specific adaptations.

The integration of SR with object detection frameworks opens several avenues for future exploration. Lightweight SR architectures, designed to operate on edge devices, could make real-time processing feasible for UAVs and other remote sensing platforms. Additionally, domain-specific SR models tailored to satellite imagery could further enhance detection accuracy, particularly for specialized applications such as environmental monitoring or military reconnaissance.

The study demonstrates that super-resolution techniques serve as a powerful preprocessing tool to enhance object detection in satellite imagery. By bridging the gap between coarse input data and high-precision detection models, SR significantly improves the reliability and effectiveness of remote sensing systems. As SR technologies evolve, their integration with advanced detection frameworks like YOLO promises to unlock new capabilities in satellite imagery analysis, benefiting a wide range of applications in science, urban development, and defense.

2.3 Object Detection in 20 Years: A Survey

The paper, *Object Detection in 20 Years: A Survey* provides a comprehensive review of the evolution of object detection techniques, spanning from the 1990s to 2022. It highlights key milestones in the field, categorizing developments into two eras: traditional methods relying on handcrafted features, and modern approaches driven by deep learning. The survey covers various components of object detection systems, including datasets, evaluation metrics, and technical innovations.

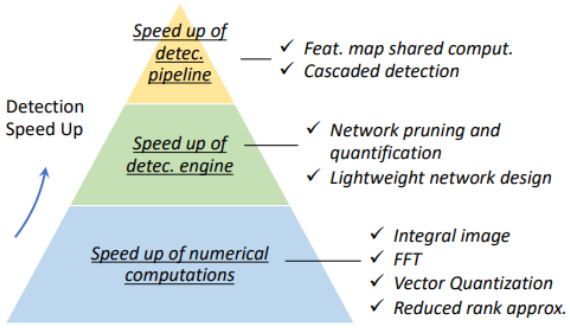

Figure 3: An overview of the speed-up techniques in object detection.

Traditional methods, such as the Viola-Jones detector, the Histogram of Oriented Gradients (HOG) and the Deformable Part-based Models (DPM), relied on sliding windows and handcrafted features for object representation. These methods were foundational, but limited by computational power and accuracy.

The deep learning era, initiated by the success of convolutional neural networks (CNNs), brought transformative changes. The introduction of R-CNN and its successors, such as Fast R-CNN, Faster R-CNN, and YOLO, integrated end-to-end learning and improved detection accuracy and speed. Techniques like feature pyramids, anchor-based and anchor-free approaches, and attention mechanisms have further advanced the field.

The survey also examines state-of-the-art methods, such as DETR and transformer-based models, which have shifted the paradigm towards global context reasoning and end-to-end optimization. The chapter discusses challenges such as small object detection, dense scenes, and real-time performance, as well as innovations in speed-up techniques, including network pruning, lightweight models, and hardware optimizations.

In general, the paper not only chronicles the rapid advances in object detection but also identifies trends and future research directions, such as more efficient models, improved generalization across datasets, and applications in emerging fields.

2.4 You Only Look Once: Unified, Real-Time Object Detection

The paper *You Only Look Once: Unified, Real-Time Object Detection* introduces YOLO, a novel approach to object detection that reframes the task as a single regression problem, predicting both bounding boxes and class probabilities in one evaluation. Unlike traditional methods like R-CNN and sliding window approaches, YOLO utilizes a single convolutional neural network that processes the entire image during training and testing, incorporating global context and optimizing the detection pipeline end-to-end.

YOLO divides the image into an S×SS times SS×S grid, where each grid cell predicts bounding boxes, confidence scores, and conditional class probabilities. This design simplifies the detection process, enabling real-time performance of up to 45 frames per second (fps) with the base model and 155 fps with the smaller Fast YOLO model. YOLO achieves this speed without compromising detection performance, achieving a mean average precision (mAP) of 63.4% on the PASCAL VOC dataset, which is competitive with state-of-the-art detectors.

Figure 4: An overview of the speed-up techniques in object detection.

The model is highly generalizable, outperforming other methods like R-CNN and DPM when applied to non-natural images such as artwork, due to its ability to reason about the spatial and contextual relationships of objects. However, YOLO makes more localization

errors and struggles with detecting smaller objects or objects in close proximity due to its strong spatial constraints and relatively coarse features.

Despite these limitations, YOLO's unified framework reduces false positives on background regions and simplifies the detection pipeline by eliminating complex components like region proposals or sliding windows. This makes YOLO a groundbreaking model for real-time applications in diverse fields, offering a tradeoff between speed, simplicity, and accuracy that redefines the standard for object detection systems.

2.5 Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks

The paper, *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks* by Ren et al., introduced a groundbreaking approach to object detection by integrating the region proposal generation and detection tasks into a single, unified framework. This integration was achieved through the development of Region Proposal Networks (RPNs), which significantly addressed the computational inefficiencies of earlier methods like R-CNN and Fast R-CNN. By sharing convolutional layers between the proposal generation and detection tasks, RPNs reduced the redundancy in feature computation, enabling faster and more efficient object detection.

The RPN component generates region proposals by sliding a small network over the convolutional feature map of an image. This network predicts object bounds and objectness scores at multiple scales and aspect ratios using a mechanism called anchors. Anchors serve as reference boxes that adapt to various object sizes and shapes, making the framework robust in detecting objects of different dimensions. These proposals are refined through a series of bounding box regressions and classifications, resulting in high-quality region proposals that can be directly fed into the detection network.

Another key innovation in Faster R-CNN is its ability to achieve near real-time performance without compromising detection accuracy. The network's unified structure eliminates the need for external proposal methods, such as Selective Search, which were computationally expensive and limited the real-time applicability of previous approaches. Faster R-CNN demonstrated its effectiveness on benchmark datasets like PASCAL VOC and MS COCO, achieving state-of-the-art accuracy while maintaining faster inference times compared to its predecessors.

The architecture of Faster R-CNN is designed to be modular, allowing the use of different backbone networks, such as VGG, ResNet, or Inception, to extract features from images. This flexibility enables the framework to adapt to various datasets and tasks, making it a versatile choice for object detection in diverse scenarios. The approach also incorporates multi-task learning, where the shared features are optimized simultaneously for region proposal generation and object classification, further enhancing its efficiency.

Faster R-CNN's advancements have made it particularly suited for applications requiring precision and speed, such as satellite imagery analysis. In these scenarios, the ability to detect small, occluded, or densely packed objects is critical. The use of anchors and shared convolutional features ensures that even fine details in high-resolution images are effectively captured. This capability is essential for military satellite imagery, where identifying small or camouflaged military vehicles amidst complex backgrounds poses a significant challenge.

In addition to its technical innovations, Faster R-CNN set a foundation for subsequent advancements in object detection. Its introduction of RPNs has inspired the development of newer models that build upon this concept, such as Mask R-CNN and Cascade R-CNN, extending its impact to tasks like instance segmentation and multi-stage detection. Faster R-CNN remains a cornerstone in the evolution of object detection techniques, offering a balance between speed, accuracy, and flexibility.

Overall, Faster R-CNN has proven to be a transformative approach in the field of object detection. Its innovations in proposal generation, unified architecture, and adaptability have established it as a benchmark model. For tasks like satellite imagery analysis, its ability to efficiently handle high-resolution data and detect small or complex objects makes it an invaluable tool, particularly when combined with super-resolution techniques to enhance input quality.

3 Methods and Experiments

After reviewing the above papers, we decided that Super Resolution using CycleGAN would be effective for our end goal of improving the performance of object detection. So this time, we will configure our code based on MCGR to test how CycleGAN is effective for SR and move on to see how it works.

3.1 Dataset

The xView dataset is a high-resolution satellite imagery dataset specifically designed for advanced object detection tasks. With a ground sampling distance (GSD) of 0.3 meters, it offers significantly higher resolution compared to most publicly available satellite imagery datasets, providing detailed visual information essential for accurate object recognition.

Figure 5: xView data image example

The dataset encompasses over 1,400 square kilometers of imagery and includes more than one million annotated objects spanning 60 distinct classes. These classes cover a diverse array of object types, including military vehicles, helicopters, naval ships, and other critical military assets. Such diversity makes xView an optimal choice for research focused on the detection and classification of military vehicles and related objects.

3.1.1 *Key Features of xView.*

One of the most significant aspects of the xView dataset is its exceptionally high resolution, made possible by data collected from the WorldView-3 satellite. This satellite captures images with a ground sampling distance of 0.3 meters, making it possible to detect fine details that are often missed in lower-resolution datasets. This level of detail is critical for identifying small or complex objects in satellite imagery.

Another important feature is the dataset's diversity in object classes. With over 60 distinct categories, xView provides a rich collection of annotated data that supports a variety of detection tasks. These categories include objects such as military vehicles, naval ships, and helicopters, all of which are meticulously labeled to enable high-performance training and evaluation of object detection models.

The data set also stands out because of its scale. It contains more than one million bounding box annotations in more than 1,400 square kilometers of satellite imagery. This sheer volume of annotations ensures that machine learning models trained on xView can achieve robust performance and generalize effectively to new scenarios.

In addition to its detailed annotations, xView is designed to work seamlessly with modern machine learning frameworks. For example, it is officially supported by the YOLO (You Only Look Once) object detection framework, making it easy for researchers to incorporate it into their workflows. It also integrates with TensorFlow's object detection API, allowing users to leverage pre-trained models and apply transfer learning techniques.

Lastly, the xView dataset is highly suited for overhead object detection tasks, a key area in satellite imagery analysis. Its combination of high-resolution data and diverse object annotations provides researchers and practitioners with a valuable resource for developing and testing advanced detection algorithms.

3.1.2 *Applications and Research Implications.*

The xView dataset has numerous applications in the field of object detection and satellite imagery analysis. One of its primary use cases is in military asset monitoring, where it helps in identifying and tracking objects such as vehicles, ships, and aircraft. This capability is particularly valuable for defense and intelligence operations, where timely and accurate detection of military assets can have significant strategic implications.

Beyond military applications, xView is also used in urban planning. The dataset's high resolution and detailed annotations enable the identification of infrastructure elements, such as buildings, roads, and vehicles, facilitating informed decision-making in urban development projects.

Another important application is disaster response. The ability to detect and classify objects in satellite imagery is crucial during natural disasters or emergencies, where rapid assessment of the affected area can save lives. By providing detailed object annotations, xView enables models to detect key objects in disaster-stricken regions, aiding rescue and relief efforts.

From a research perspective, xView is instrumental in advancing the field of computer vision. Its high-resolution imagery and extensive object diversity present unique challenges and opportunities for developing state-of-the-art detection algorithms. Furthermore, its compatibility with YOLO and other frameworks highlights its role in driving innovation in machine learning techniques for satellite image analysis. Overall, the xView dataset is a cornerstone resource for researchers and practitioners aiming to address complex problems in object detection and remote sensing.

3.1.3 *Processing xView Data for YOLO Framework.*

To utilize the xView dataset for training and validating object detection models in the YOLO framework, a structured preprocessing pipeline was implemented. The xView dataset, originally provided in GeoTIFF format with accompanying GeoJSON labels, was converted and reorganized to adhere to the YOLO-compatible directory and annotation structure. This section details the steps involved in preparing the dataset.

Initial Dataset Structure. The xView dataset contains high-resolution satellite imagery split into training and validation subsets under the directories `train_images` and `val_images`. It includes 847 images in total, along with a corresponding GeoJSON file, `xView_train.geojson`, which contains bounding box annotations for various object classes.

Reorganization of Image Data. First, all images were consolidated into a single directory named `images`. This was achieved by programmatically moving the contents of `train_images` and `val_images` into the new directory while preserving their respective subsets under subdirectories named `train` and `val`. The following Python script demonstrates the process:

```
# Move images
images = Path(dir / 'images')
images.mkdir(parents=True, exist_ok=True)
Path(dir / 'train_images').rename(dir / 'images' /
Path(dir / 'val_images').rename(dir / 'images' / 'val')

# Split
autosplit(dir / 'images' / 'train')
```

Figure 6: Python code for reorganization of xView dataset images.

To ensure compatibility with YOLO training requirements, the `autosplit` function was utilized to divide the images in the `train` directory into training (90%) and validation (10%) subsets. The resulting split generated `autosplit_train.txt` and `autosplit_val.txt` files, listing the respective image paths. These files enable seamless loading of images during the training process.

Annotation Conversion. The xView dataset's GeoJSON annotations were converted into YOLO's text-based label format. Each annotation was processed to extract bounding box coordinates and class labels, which were normalized to the YOLO format. The YOLO label file for each image contains lines corresponding to each object, with the format:

`<class_id> <x_center> <y_center> <width> <height>`

Where the coordinates and dimensions are normalized relative to the image size.

```
└── datasets
    └── xView  ← downloads here (20.7 GB)
        └── train_images  ← .tif dir of train datasets
        └── val_images    ← .tif dir of valid datasets
        └── xView_train.geojson ← label
```

Figure 7: Original xView dataset structure before conversion.

The converted label files were organized under a directory named `labels`, with separate subdirectories for `train` and `val` annotations, corresponding to their respective image subsets.

Final Dataset Structure. After preprocessing, the dataset was organized as follows:

```
└── datasets
    └── xView  ← downloads here (20.7 GB)
        └── images
            └── train  ← .tif dir of train datasets
            └── val    ← .tif dir of valid datasets
            └── autosplit_train.txt  ← train images 90% of 847 train images
            └── autosplit_val.txt    ← train images 10% of 847 train images
        └── labels
            └── train  ← .txt dir of train labels
        └── xView_train.geojson ← original labels
```

Figure 8: Final xView dataset structure compatible with YOLO framework.

Conclusion. This preprocessing pipeline effectively transforms the xView dataset into a format compatible with the YOLO framework. By reorganizing images, generating YOLO-compatible labels, and ensuring a structured directory hierarchy, this process streamlines the integration of the xView dataset for object detection tasks. The resulting dataset structure facilitates efficient training and evaluation of YOLO models, enabling enhanced performance on high-resolution satellite imagery.

3.2 SR

3.2.1 *EDSR.*

```python
def res_block(x_in, filters, scaling):
    x = Conv2D(filters, 3, padding='same', activation='relu')(x_in)
    x = Conv2D(filters, 3, padding='same')(x)
    if scaling:
        x = Lambda(lambda t: t * scaling)(x)
    x = Add()([x_in, x])
    return x
```

Figure 9: Residual Block Code for EDSR

The Enhanced Deep Super-Resolution Network (EDSR) is an enhanced super-resolution model designed on top of SRResNet, which significantly improves the accuracy and efficiency in super-resolution tasks. EDSR shows good results in super-resolution performance based on peak signal-to-noise ratio (PSNR), and is expected to be particularly strong in satellite imagery and military applications where high resolution is required. EDSR is characterized by the fact that the Residual Network improves the structure of SRResNet, solving the problem of gradient decay and stabilizing learning through residual connections. It also removes the Batch Normalization Layer to reduce memory usage and optimize performance, and is designed to include more layers than existing models to learn complex patterns. EDSR is advantageous for large-scale high-resolution image processing and has a modular design that provides scalability to handle different resolution scales. EDSR is considered the industry standard for static image super-resolution, and the authors believe that it is suitable for restoring complex details in military satellite imagery, so they selected EDSR, which is expected to perform reliably on military data with a high PSNR target among CNN-based models, for the experiments in this paper.

The code in Figure 9 defines a function res_block which constructs a residual block used within the EDSR (Enhanced Deep Residual Network) model.

The res_block function is designed to create a residual block, a fundamental component of deep residual networks. This block takes three parameters: x_in (the input tensor), filters (the number of convolutional filters), and scaling (an optional scaling factor for the block's output).

The function begins by applying a convolutional layer with the specified number of filters (e.g., 64) and a 3x3 kernel to the input tensor x_in. The convolution uses 'same' padding to ensure that the spatial dimensions of the output match the input. An activation function, ReLU, is applied to introduce non-linearity into the model.

Next, another convolutional layer is applied with the same number of filters and kernel size, again using 'same' padding. This layer does not use an activation function, allowing the network to learn a more flexible mapping.

If a scaling factor is provided, the output of the second convolutional layer is scaled by this factor using a Lambda layer. This step can help stabilize the training of very deep networks by preventing the output values from becoming too large.

After the (optional) scaling, the function adds the original input tensor x_in to the output of the second convolutional layer using a shortcut connection. This addition implements the residual learning mechanism, enabling the network to learn residual mappings rather than direct mappings. This approach helps mitigate issues such as the vanishing gradient problem, making it easier to train deeper networks.

Finally, the function returns the output tensor x, which now includes the learned residual information. This residual block can then be stacked multiple times within the network to build a deeper model.

The code in Figure 10 defines a function upsample which is used to increase the spatial resolution of feature maps within the EDSR (Enhanced Deep Residual Network) model.

The upsample function takes three parameters: x (the input tensor), scale (the factor by which the spatial resolution should be increased), and num_filters (the number of filters to use in the convolutional layers).

Inside the upsample function, there is a nested helper function called upsample_1. This helper function performs a single step of upsampling. It accepts three parameters: x (the input tensor),

```python
def upsample(x, scale, num_filters):
    def upsample_1(x, factor, **kwargs):
        x = Conv2D(num_filters * (factor ** 2), 3, padding='same', **kwargs)(x)
        return Lambda(pixel_shuffle(scale=factor))(x)

    if scale == 2:
        x = upsample_1(x, 2, name='conv2d_1_scale_2')
    elif scale == 3:
        x = upsample_1(x, 3, name='conv2d_1_scale_3')
    elif scale == 4:
        x = upsample_1(x, 2, name='conv2d_1_scale_2')
        x = upsample_1(x, 2, name='conv2d_2_scale_2')

    return x
```

Figure 10: Upsampling Code for EDSR

factor (the upsampling factor for this particular step), and additional keyword arguments (**kwargs).

In upsample_1, a convolutional layer is applied to the input tensor x using num_filters * (factor ** 2) filters with a kernel size of 3x3 and 'same' padding. The number of filters is scaled by factor ** 2 to prepare for the pixel shuffle operation, ensuring the correct number of channels for the subsequent reshaping. After the convolution, a Lambda layer applies the pixel_shuffle operation, which rearranges the elements of the convolutional output tensor to increase the spatial resolution by the specified factor.

Returning to the main upsample function, it handles different cases for the scale parameter. If the scale is 2, it calls upsample_1 once with a factor of 2. If the scale is 3, it calls upsample_1 once with a factor of 3. If the scale is 4, it calls upsample_1 twice, each time with a factor of 2. This effectively achieves a 4x upscaling by performing two consecutive 2x upsampling operations. Each call to upsample_1 includes a unique layer name for easier identification.

Finally, the upsample function returns the upsampled tensor x.

3.2.2 SRGAN.

Super-Resolution Generative Adversarial Network (SRGAN) is a super-resolution model based on Generative Adversarial Network (GAN) that focuses on providing high visual quality. Unlike PSNR-based models, SRGAN's strengths lie in restoring fine texture and realistic detail in images. SRGAN's Generator takes a low-resolution image as input and produces a high-resolution image, and is designed based on a residual network. SRGAN's Discriminator compares the generated image with the actual high-resolution image to guide the generator to produce a more realistic image, and Perceptual Loss, a VGG-based loss function, calculates the loss based on a high-level feature map extracted from the VGG network instead of simple pixel loss to produce a visually natural image, and then goes through a process of adversarial training to learn more realistic and high-quality images through competition between the generator and discriminator. SRGAN is characterized by its excellent performance on the Mean Opinion Score (MOS), which is a human visual evaluation rather than PSNR. This makes it a good model for situations where visual clarity and natural details are important.

The code in Figure 11 defines a residual block function (‘res_block‘) used in the SRGAN (Super-Resolution Generative Adversarial Network) model. This block is crucial for enhancing image quality in super-resolution tasks.

```python
def res_block(x_in, num_filters, momentum=0.8):
    x = Conv2D(num_filters, kernel_size=3, padding='same')(x_in)
    x = BatchNormalization(momentum=momentum)(x)
    x = PReLU(shared_axes=[1, 2])(x)
    x = Conv2D(num_filters, kernel_size=3, padding='same')(x)
    x = BatchNormalization(momentum=momentum)(x)
    x = Add()([x_in, x])
    return x
```

Figure 11: Residual Block Code for SRGAN

The ‘res_block‘ function constructs a residual block, an essential component in SRGAN for efficient feature learning and preserving information across layers. It takes three parameters: ‘x_in‘ (the input tensor), ‘num_filters‘ (the number of filters in the convolutional layers), and ‘momentum‘ (used for batch normalization, with a default value of 0.8).

The function begins by applying a convolutional layer to the input tensor ‘x_in‘. This layer uses ‘num_filters‘ filters with a kernel size of 3x3 and 'same' padding, ensuring the output has the same spatial dimensions as the input. The output of this convolution is then passed through a batch normalization layer, which helps stabilize and speed up the training process by normalizing the output of the convolutional layer. The ‘momentum‘ parameter controls the moving average of batch statistics.

Next, a Parametric ReLU (PReLU) activation function is applied. PReLU introduces non-linearity into the model and helps learn the activation parameters during training. The ‘shared_axes‘ parameter specifies which axes to share the parameters across, in this case, the spatial dimensions (1 and 2).

Following the activation, another convolutional layer is applied with the same number of filters, kernel size, and padding as the first one. This layer further processes the features extracted from the input tensor. The output of this convolution is again passed through a batch normalization layer with the specified ‘momentum‘.

Finally, the original input tensor ‘x_in‘ is added to the output of the second batch normalization layer using a shortcut connection. This addition implements the residual learning mechanism, allowing the network to learn residual functions, which improves the training efficiency and effectiveness, particularly in deep networks.

The function then returns the final output tensor ‘x‘, which now contains the learned residual information. This residual block can be stacked multiple times within the SRGAN model to build a deeper and more powerful network for super-resolution tasks.

The code in Figure 12 defines the generator function used in the SRGAN (Super-Resolution Generative Adversarial Network) model. This generator is responsible for creating high-resolution images from low-resolution inputs.

The ‘generator‘ function constructs a deep convolutional neural network designed to upscale images. It takes two parameters: ‘num_filters‘ (the number of filters to use in the convolutional layers, defaulting to 64) and ‘num_res_blocks‘ (the number of residual blocks to include, defaulting to 16).

The function starts by defining the input tensor ‘x_in‘, which is expected to be a three-channel (RGB) image of variable height and width. The input is then normalized using a ‘Lambda‘ layer,

```python
def generator(num_filters=64, num_res_blocks=16):
    x_in = Input(shape=(None, None, 3))
    x = Lambda(normalize_01)(x_in)

    x = Conv2D(num_filters, kernel_size=9, padding='same')(x)
    x = x_1 = PReLU(shared_axes=[1, 2])(x)

    for _ in range(num_res_blocks):
        x = res_block(x, num_filters)

    x = Conv2D(num_filters, kernel_size=3, padding='same')(x)
    x = BatchNormalization()(x)
    x = Add()([x_1, x])

    x = upsample(x, num_filters * 4)
    x = upsample(x, num_filters * 4)

    x = Conv2D(3, kernel_size=9, padding='same', activation='tanh')(x)
    x = Lambda(denormalize_m11)(x)

    return Model(x_in, x)
```

Figure 12: Generator Code for SRGAN

which scales the pixel values to a range suitable for processing by the network.

The first layer of the network is a convolutional layer with 'num_filters' filters, a large kernel size of 9x9, and 'same' padding. This layer captures high-level features from the input image. The output of this layer is then passed through a PReLU (Parametric ReLU) activation function, which introduces non-linearity and helps the network learn more complex representations. The 'shared_axes' parameter ensures the PReLU parameters are shared across the spatial dimensions.

Next, the network enters a loop where it applies a series of residual blocks. Each residual block, defined by the 'res_block' function, consists of two convolutional layers with batch normalization and PReLU activations, along with a shortcut connection that adds the input of the block to its output. This series of residual blocks allows the network to learn residual mappings, improving its ability to reconstruct high-frequency details in the images.

After passing through the residual blocks, another convolutional layer is applied with the same number of filters and a kernel size of 3x3. This layer is followed by batch normalization. The output of this layer is added to the output of the initial convolutional layer using a shortcut connection. This addition helps preserve the high-level features captured by the initial layers.

The next step involves upsampling the feature maps to increase their spatial resolution. The 'upsample' function is called twice, each time increasing the resolution by a factor of 2. Inside the 'upsample' function, a convolutional layer with a large number of filters ('num_filters * 4') is applied, followed by a pixel shuffle operation that rearranges the elements of the tensor to increase its spatial dimensions.

Finally, a convolutional layer with 3 filters (corresponding to the RGB channels) and a large kernel size of 9x9 is applied. This layer uses a 'tanh' activation function to ensure the output pixel values are scaled between -1 and 1. The output is then denormalized using another 'Lambda' layer to bring the pixel values back to their original range.

The function concludes by creating and returning a 'Model' object that connects the input tensor 'x_in' to the final output tensor 'x'. This model represents the generator network of the SRGAN, capable of transforming low-resolution images into high-resolution ones by learning complex mappings through deep convolutional layers and residual blocks.

```python
def discriminator_block(x_in, num_filters, strides=1, batchnorm=True, momentum=0.8):
    x = Conv2D(num_filters, kernel_size=3, strides=strides, padding='same')(x_in)
    if batchnorm:
        x = BatchNormalization(momentum=momentum)(x)
    return LeakyReLU(alpha=0.2)(x)
```

Figure 13: Discriminator Block Code for SRGAN

The code in Figure 13 defines a discriminator_block function used in the SRGAN (Super-Resolution Generative Adversarial Network) model. This block is a key component of the discriminator, which is responsible for distinguishing between real and generated high-resolution images.

The discriminator_block function constructs a convolutional block used in the discriminator network. It takes five parameters: x_in (the input tensor), num_filters (the number of filters for the convolutional layer), strides (the stride length for the convolution, defaulting to 1), batchnorm (a boolean indicating whether to apply batch normalization, defaulting to True), and momentum (the momentum parameter for batch normalization, defaulting to 0.8).

The function begins by applying a convolutional layer to the input tensor x_in. This convolutional layer uses num_filters filters with a kernel size of 3x3 and 'same' padding, ensuring that the spatial dimensions of the output match those of the input. The strides parameter allows the layer to reduce the spatial dimensions of the output if necessary.

If batchnorm is set to True, the function then applies a batch normalization layer to the output of the convolutional layer. Batch normalization helps stabilize and speed up the training process by normalizing the output of the convolutional layer. The momentum parameter controls the moving average of batch statistics, ensuring that the normalization is based on recent batch data.

Following batch normalization (if applied), the function applies a Leaky ReLU activation function with an alpha parameter of 0.2. Leaky ReLU introduces non-linearity into the model and allows a small gradient when the unit is not active, which helps prevent the dying ReLU problem.

The function finally returns the output tensor x, which now contains the processed features. This tensor can then be passed to subsequent layers or blocks in the discriminator network.

The code in Figure 14 defines the discriminator function used in the SRGAN (Super-Resolution Generative Adversarial Network) model. The discriminator's role is to distinguish between real high-resolution images and those generated by the generator.

The discriminator function constructs a deep convolutional neural network designed to classify images as either real or fake. It starts by defining the input tensor 'x_in', which is expected to be a three-channel (RGB) image of a specified size, 'HR_SIZE x HR_SIZE'. The input is normalized using a 'Lambda' layer, which

```python
def discriminator(num_filters=64):
    x_in = Input(shape=(HR_SIZE, HR_SIZE, 3))
    x = Lambda(normalize_m11)(x_in)

    x = discriminator_block(x, num_filters, batchnorm=False)
    x = discriminator_block(x, num_filters, strides=2)

    x = discriminator_block(x, num_filters * 2)
    x = discriminator_block(x, num_filters * 2, strides=2)

    x = discriminator_block(x, num_filters * 4)
    x = discriminator_block(x, num_filters * 4, strides=2)

    x = discriminator_block(x, num_filters * 8)
    x = discriminator_block(x, num_filters * 8, strides=2)

    x = Flatten()(x)

    x = Dense(1024)(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dense(1, activation='sigmoid')(x)

    return Model(x_in, x)
```

Figure 14: Discriminator Code for SRGAN

scales the pixel values to a suitable range for processing by the network.

The network consists of several convolutional blocks, each defined by the ‘discriminator_block‘ function. These blocks include convolutional layers, optional batch normalization, and Leaky ReLU activation functions. The first block applies a convolutional layer with the specified number of filters and disables batch normalization. This block uses ’same’ padding to preserve the input dimensions and a stride of 1.

The second block, like the first, uses the same number of filters but increases the stride to 2, effectively downsampling the feature maps. The subsequent blocks follow a similar pattern but progressively increase the number of filters. The third and fourth blocks double the number of filters to ‘num_filters * 2‘ and apply convolutional layers twice, each time using a stride of 2 to downsample the feature maps.

The fifth and sixth blocks continue this pattern, doubling the number of filters again to ‘num_filters * 4‘ and applying convolutional layers with strides of 2. Finally, the seventh and eighth blocks double the number of filters one last time to ‘num_filters * 8‘, continuing the pattern of downsampling and feature extraction.

After passing through the convolutional blocks, the feature maps are flattened into a single dimension using the ‘Flatten‘ layer. This prepares the features for the fully connected layers. The first dense (fully connected) layer has 1024 units and is followed by a Leaky ReLU activation function with an alpha value of 0.2. This layer helps in learning complex features from the flattened feature maps.

The final dense layer has a single unit with a sigmoid activation function, which outputs a probability indicating whether the input image is real (closer to 1) or generated (closer to 0). The function concludes by creating and returning a ‘Model‘ object that connects the input tensor ‘x_in‘ to the final output tensor ‘x‘. This model represents the discriminator network of the SRGAN, capable of classifying high-resolution images as real or fake.

3.2.3 *CycleGAN.*

CycleGAN is a GAN-based model that learns to transform images between different domains. It provides a unique approach that breaks away from the traditional methods of super-resolution work, allowing images to be converted to high resolution while preserving their style or character. Cycle GAN uses the Cycle Consistent Loss function. Cycle Consistency Loss is a loss function that learns to be as similar as possible to the original image when the transformed image is transformed back into the original domain. CycleGAN belongs to Unsupervised Learning, which means it is slow to learn without pairwise aligned data, which is advantageous for satellite imagery where aligned high-resolution-low-resolution datasets are scarce. It also uses two GAN networks, two generators and two descriptors to perform interconversion between the two domains. CycleGAN is capable of super-resolution conversion while maintaining the structural characteristics of the image, such as the outline of a vehicle or the texture of an object. In other words, CycleGAN is a model that can increase resolution while maintaining visual characteristics. With Cycle GAN, when features differ significantly between domains, such as the presence of camouflage patterns in certain areas of military satellite imagery, or when images are taken in specialized environments, CycleGAN can preserve these features and convert them into high-resolution images. In particular, it can learn effectively from limited datasets.

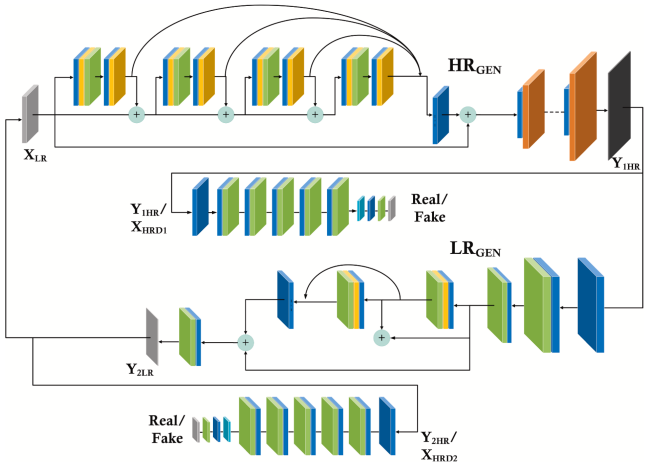

Figure 15: CycleGAN Model Graphic

The code defines a residual block function, GAN_res_block, used in the CycleGAN model. This block is crucial for maintaining the identity of the input while allowing the network to learn complex transformations.

The GAN_res_block function constructs a residual block, a fundamental component of the CycleGAN architecture. It takes three parameters: x_in (the input tensor), num_filters (the number of filters in the convolutional layers), and skip_connection (a boolean that determines whether to use a skip connection, defaulting to True).

```python
def GAN_res_block(x_in, num_filters, skip_connection=True):
    x = Conv2D(num_filters, kernel_size=3, padding='same')(x_in)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    x = Conv2D(num_filters, kernel_size=3, padding='same')(x)
    x = BatchNormalization()(x)
    if skip_connection == True:
        x = Add()([x_in, x])
    return x
```

Figure 16: HR Generator's Residual Block Code for CycleGAN

The function starts by applying a convolutional layer to the input tensor x_in. This convolutional layer uses num_filters filters with a kernel size of 3x3 and 'same' padding, ensuring the output has the same spatial dimensions as the input. The output of this layer is then passed through a batch normalization layer, which helps stabilize and speed up the training process by normalizing the output of the convolutional layer. A ReLU activation function is applied next, introducing non-linearity into the model and allowing it to learn more complex representations.

Following the activation, another convolutional layer is applied with the same number of filters and kernel size, again using 'same' padding. This layer further processes the features extracted from the input tensor. The output of this convolution is again passed through a batch normalization layer, ensuring that the normalization is based on the recent batch data.

If the skip_connection parameter is set to True, the function adds the original input tensor x_in to the output of the second batch normalization layer using a shortcut connection. This addition implements the residual learning mechanism, enabling the network to learn residual functions rather than direct mappings. This approach helps mitigate issues such as the vanishing gradient problem, making it easier to train deeper networks and preserving the identity of the input.

Finally, the function returns the output tensor x, which now contains the learned residual information. This residual block can be stacked multiple times within the CycleGAN model to build a deeper and more powerful network capable of learning complex image-to-image transformations while maintaining input identity.

The code in Figure 17 defines the HR_generator function used in the CycleGAN model, which aims to generate high-resolution images from low-resolution inputs.

The HR_generator function constructs a deep convolutional neural network designed to upscale images by a specified factor (scale). The network uses several residual blocks to learn complex transformations while maintaining the identity of the input images. The function takes three parameters: scale (the upscaling factor, defaulting to 4), num_filters (the number of filters in the convolutional layers, defaulting to 64), and num_res_blocks (the number of residual blocks, defaulting to 3).

The function starts by defining the input tensor x_in, which is expected to be a three-channel (RGB) image of variable height and width. The input is first processed by a convolutional layer (x2)

```python
def HR_generator(scale=4, num_filters=64, num_res_blocks=3):
    x_in = Input(shape=(None, None, 3))
    x = x2 = Conv2D(num_filters, 3, padding='same')(x_in)

    d_x = []
    for _ in range(num_res_blocks):
        x = GAN_res_block(x, num_filters)
        d_x.append(x)

    x = GAN_res_block(x, num_filters, skip_connection=False)

    d_x.append(x)
    x = Concatenate()(d_x)

    x = Conv2D(num_filters, kernel_size=1, padding='same')(x)
    x = Add()([x2, x])

    x = upsample(x, scale, num_filters)
    x = Conv2D(3, 3, padding='same')(x)

    return Model(x_in, x, name="gan_HR")
```

Figure 17: HR Generator Code for CycleGAN

with a kernel size of 3x3 and 'same' padding, using num_filters filters. This layer extracts initial features from the input image.

Next, an empty list d_x is initialized to store the outputs of the residual blocks. The network then enters a loop where it applies a series of residual blocks (GAN_res_block) to the input tensor x. Each residual block consists of two convolutional layers with batch normalization and ReLU activation, and optionally includes a skip connection that adds the input of the block to its output. The output of each residual block is appended to the d_x list.

After passing through the specified number of residual blocks, another residual block is applied to the tensor x with the skip connection disabled (skip_connection=False). This ensures that the last residual block's output is directly used for further processing. The output of this block is also appended to the d_x list.

The concatenated outputs of all residual blocks stored in d_x are then merged using a Concatenate layer, which combines these features along the channel dimension. This merged output is processed by another convolutional layer with a kernel size of 1x1 and 'same' padding, which helps in reducing the dimensionality and combining the features. The result is then added to the output of the initial convolutional layer (x2) using a shortcut connection.

The next step involves upsampling the feature maps to increase their spatial resolution by the specified scale factor. The upsample function is called with the appropriate parameters to perform this task. Inside the upsample function, convolutional layers and the pixel shuffle operation are used to rearrange the elements of the tensor, effectively increasing its spatial dimensions.

Finally, a convolutional layer with 3 filters and a kernel size of 3x3 is applied to the upsampled tensor. This layer outputs the final high-resolution image with three color channels (RGB). The function concludes by creating and returning a Model object that connects the input tensor x_in to the final output tensor x. This model represents the high-resolution generator network of the CycleGAN, capable of transforming low-resolution images into high-resolution ones.

```python
def LR_generator(num_filters=64):
    x_in = Input(shape=(None, None, 3))

    x = Conv2D(num_filters, 3, strides = 2, padding='same', activation='relu')(x_in) # downscaling
    x = Conv2D(num_filters, 3, strides = 2, padding='same', activation='relu')(x) # downscaling
    skip1 = x

    x = Conv2D(num_filters, kernel_size=3, padding='same')(x)
    x = BatchNormalization(momentum=0.8)(x)
    x = ReLU()(x)
    skip2 = x

    x = Conv2D(num_filters, kernel_size=3, padding='same')(x)
    x = BatchNormalization(momentum=0.8)(x)
    x = ReLU()(x)

    x = Concatenate()([x, skip2])
    x = Conv2D(num_filters, kernel_size=3, padding='same')(x)

    x = Add()([skip1, x])

    x = Conv2D(3, 3, padding='same', activation='relu')(x)

    return Model(x_in, x)
```

Figure 18: LR Generator Code for CycleGAN

The code in Figure 18 defines the LR_generator function used in the CycleGAN model, which is responsible for generating low-resolution images from high-resolution inputs.

The LR_generator function constructs a convolutional neural network designed to downscale images. It begins by defining the input tensor x_in, which is expected to be a three-channel (RGB) image of variable height and width. The first two layers of the network are convolutional layers with a kernel size of 3x3, num_filters (defaulting to 64), and 'same' padding. Both layers use a stride of 2, which effectively downsamples the spatial dimensions of the input image by a factor of 2 with each layer. The ReLU activation function is applied to both layers to introduce non-linearity. The outputs of these layers are saved as skip1 and skip2 for later use.

Following the initial downsampling layers, the network applies another convolutional layer with the same number of filters and kernel size, but with a stride of 1 to preserve the spatial dimensions. This layer is followed by batch normalization with a momentum of 0.8 to stabilize and speed up the training process. The ReLU activation function is then applied.

The network then applies a second set of convolutional and batch normalization layers, similar to the previous set, but without downsampling (stride of 1). This ensures that the spatial dimensions remain unchanged while allowing the network to learn more complex features. The output of this layer is saved as skip2.

The outputs from the initial downsampling layers (skip1 and skip2) and the subsequent convolutional layers are concatenated along the channel dimension using a Concatenate layer. This concatenation combines the features learned at different stages of the network.

Next, the concatenated tensor is passed through another convolutional layer with the same number of filters and a kernel size of 3x3, followed by another convolutional layer with a kernel size of 3x3. Both layers use 'same' padding to maintain the spatial dimensions and a ReLU activation function. The output of this layer is added to the concatenated tensor using a shortcut connection (Add layer), allowing the network to learn residual mappings and maintain the identity of the input.

Finally, the network applies one more convolutional layer with 3 filters and a kernel size of 3x3, using 'same' padding and a ReLU activation function. This layer produces the final low-resolution image with three color channels (RGB).

The function concludes by creating and returning a Model object that connects the input tensor x_in to the final output tensor x. This model represents the low-resolution generator network of the CycleGAN, capable of transforming high-resolution images into low-resolution ones.

```python
def discriminator_block(x_in, num_filters, strides=1):
    x = Conv2D(num_filters, kernel_size=3, strides=strides, padding='same')(x_in)
    return LeakyReLU(alpha=0.2)(x)

def discriminator(num_filters=64):
    x_in = Input(shape=(None, None, 3))

    x = Conv2D(num_filters, kernel_size=3, padding='same')(x_in)

    x = discriminator_block(x, num_filters)
    x = discriminator_block(x, num_filters)
    x = discriminator_block(x, num_filters)
    x = discriminator_block(x, num_filters)
    x = discriminator_block(x, num_filters, 2)

    x = discriminator_block(x, num_filters)

    return Model(x_in, x)
```

Figure 19: Discriminator Code for CycleGAN

The code in Figure 19 defines a discriminator_block function and a discriminator function used in the CycleGAN model. The discriminator's role is to distinguish between real and generated images, which is crucial for the adversarial training process.

The discriminator_block function is designed to build a convolutional block that forms the basic building unit of the discriminator network. It takes three parameters: x_in (the input tensor), num_filters (the number of filters for the convolutional layer), and strides (the stride length for the convolution, defaulting to 1).

The function begins by applying a convolutional layer to the input tensor x_in. This layer uses num_filters filters with a kernel size of 3x3 and 'same' padding, ensuring the spatial dimensions of the output match those of the input. The strides parameter controls the stride of the convolution, allowing for optional downsampling. The output of this layer is then passed through a Leaky ReLU activation function with an alpha value of 0.2. This activation introduces non-linearity into the model and allows a small gradient when the unit is not active, which helps prevent the dying ReLU problem.

The discriminator function constructs the full discriminator network using multiple discriminator_block instances. It takes one parameter: num_filters (the number of filters for the convolutional layers, defaulting to 64).

The function starts by defining the input tensor x_in, which is expected to be a three-channel (RGB) image of variable height and width. The input is first processed by a convolutional layer with num_filters filters, a kernel size of 3x3, and 'same' padding. This initial layer helps capture the basic features of the input image.

The network then applies a series of discriminator_block instances to the processed input tensor. The first few blocks use the default stride of 1, maintaining the spatial dimensions while progressively increasing the feature depth through the addition of more filters. The middle layers of the network continue to process the features with the same number of filters and stride.

Table 1: Frequency of Special Characters

Model(x4)	Num. of Paramameters	PSNR
EDSR	1.51M	29.47
SRGAN(Gen)	1.54M	35.81
CycleGAN(HR_Gen)	0.59M	27.08

As the network progresses, the stride is increased to 2 in one of the discriminator_block instances, effectively downsampling the feature maps and reducing their spatial dimensions. This downsampling helps the network learn more abstract features at a coarser level.

The final layers of the discriminator network continue to apply additional discriminator_block instances to further process the downsampled features, maintaining the same number of filters.

The function concludes by creating and returning a Model object that connects the input tensor x_in to the final output tensor x. This model represents the discriminator network of the CycleGAN, capable of classifying images as real or generated.

3.3 YOLO

The You Only Look Once (YOLO) framework is a transformative model in the field of object detection, designed to balance speed and accuracy effectively. YOLO has established itself as a go-to solution for real-time applications, particularly in military scenarios where rapid and precise decision-making is critical. Unlike multi-stage object detection pipelines, YOLO unifies object localization and classification into a single forward pass through a convolutional neural network, thereby significantly reducing inference time and computational complexity.

One of the defining strengths of YOLO is its ability to process images at remarkable speeds, with later versions such as YOLOv5 and YOLOv7 achieving frame rates of up to 155 frames per second. This capability is particularly advantageous in military operations where real-time intelligence is paramount. For instance, UAV reconnaissance missions and satellite imagery analysis require models that can quickly identify potential threats or targets without delays, enabling timely responses to evolving situations. YOLO's architecture ensures that real-time detection is achieved without compromising on accuracy, making it an indispensable tool in high-pressure environments.

YOLO's streamlined design allows it to run efficiently on edge devices with limited computational resources. This feature is critical for deployment in remote or resource-constrained military operations where heavy computational infrastructure may not be available. Whether deployed on drones, portable devices, or field surveillance systems, YOLO effectively utilizes available hardware to deliver reliable detection results. This efficiency is bolstered by its lightweight structure, which supports rapid deployment and adaptation to new environments.

Military applications often involve detecting objects in cluttered, densely packed, or visually complex environments. YOLO addresses these challenges with its anchor box mechanism, which enables precise localization of objects of varying sizes and aspect ratios. This capability is crucial for identifying military assets such as vehicles, aircraft, and naval ships, even when they are partially occluded or camouflaged. Furthermore, YOLO's non-maximum suppression (NMS) technique reduces overlapping bounding boxes, ensuring accurate detection without false positives, which is essential for actionable intelligence.

In this study, YOLO is augmented with super-resolution techniques to further enhance its detection capabilities. Satellite imagery, often constrained by limited resolution, benefits significantly from preprocessing through super-resolution methods like CycleGAN. By converting low-resolution images into high-resolution inputs, the combined system maximizes YOLO's detection accuracy. The integration of CycleGAN not only improves the visual clarity of images but also ensures that YOLO can effectively detect small or intricate objects that might otherwise be overlooked.

Another advantage of YOLO lies in its modularity and adaptability. The framework's open-source nature allows for extensive customization to suit specific military use cases. For example, versions like YOLOv5 and YOLOv7 introduce advanced features such as enhanced feature extraction and improved accuracy metrics, making them ideal for specialized datasets like xView. This adaptability ensures that YOLO can be fine-tuned to detect a wide range of military platforms, from ground vehicles to aerial drones, with minimal retraining effort.

YOLO has demonstrated its versatility across various military applications. For satellite and UAV imagery, it supports tasks such as perimeter monitoring, target identification, and battlefield mapping. Its ability to simultaneously detect multiple objects in a single frame streamlines intelligence gathering processes, enabling more efficient decision-making. In addition, YOLO's low false-positive rate and high accuracy provide confidence in its outputs, which is critical for mission-critical operations.

YOLO benefits from a robust community of developers and researchers who contribute to its continuous evolution. This active support network ensures that the framework remains at the forefront of object detection technology, with frequent updates and enhancements. The availability of comprehensive documentation and pretrained models further accelerates its adoption and integration into diverse applications.

In conclusion, YOLO's combination of real-time processing, high precision, hardware efficiency, and adaptability makes it a cornerstone of modern military object detection. When integrated with super-resolution techniques like CycleGAN, it achieves unparalleled performance in analyzing high-resolution satellite and UAV imagery. As military operations increasingly rely on data-driven intelligence, YOLO's ability to deliver accurate and timely detection results positions it as an essential tool for enhancing operational effectiveness and decision-making in the field.

3.3.1 YOLO training.

To illustrate the process of training a YOLO model, the code in the Figure 20 above shows the process of training a model for object detection using the Ultralytics YOLO library. In the first part of the code, we import the YOLO class using from ultralytics import YOLO, and load a pre-trained weight named "yolo11n.pt". The "n" version here is the lightweight YOLO model, which is ideal for use in resource-constrained environments. We then call

```python
from ultralytics import YOLO

model = YOLO("yolo11n.pt")

torch.cuda.empty_cache()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device)

results = model.train(data ="./xView.yaml", epochs=20, imgsz=640, verbose=True, batch=5)
```

Figure 20: YOLO training code

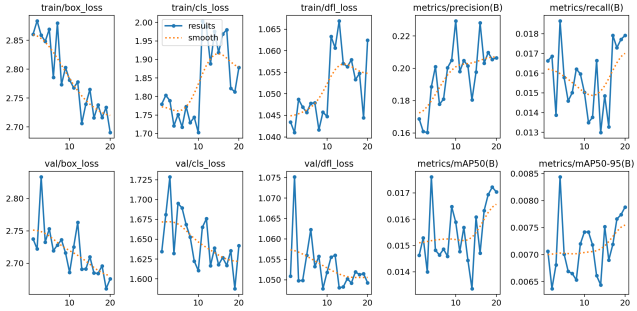

Figure 21: YOLO training result

torch.cuda.empty_cache() to empty the GPU cache and initialize it for efficient use of memory. We used torch.device to assign "cuda:0" if the GPU is available, otherwise we enable the CPU. Along the way, we place the model on the appropriate device with model.to(device). We then call model.train to start training. The main training parameters are data="./xview.yaml", which specifies the path to the dataset settings file, which defines the configuration of the training and validation datasets. epochs=20 is set to perform 20 epochs during one training session, and the image size is resized to imgsz=640. The batch size is set to batch=5 to account for GPU memory constraints. Finally, verbose=True outputs a log of the training progress. The annotations explain that the batch size was adjusted to account for GPU memory limitations, and that training is performed 20 epochs at a time, broken up to gradually complete 100 epochs. This appears to be a strategy to increase training stability and avoid memory overflow issues. At the bottom of the screen is a printout of the model's structure. This structure shows the layer-by-layer details of the YOLO model, with each layer representing the input and output connections, number of parameters, module types, and key architectural elements used. For example, ultralytics.nn.modules.conv.Conv is a convolutional layer with different sizes and strides, Concat is an operation that merges feature maps from different layers, Spatial Pyramid Pooling - Fast (SPPF) is a layer for expanding the acceptance region, and block structures like C3k2 are a lightweight feature extraction structure often used in the YOLOv5 and later model series. The entire model structure is designed to balance efficiency and performance, and the lightweight layers allow for stable operation in environments with limited hardware resources. Overall, this code demonstrates the process of training an object detection model to fit a new dataset (xview.yaml) by utilizing pretrained YOLO weights. The lightweight model structure and tuning of the training parameters to account for GPU memory is a way to maximize training efficiency in a limited resource environment, with the focus on achieving optimal performance.

To further illustrate the training process of the YOLO model, the provided training and validation curves highlight the evolution of loss metrics and performance metrics over the course of 20 epochs. The training outputs, depicted in the Figure 21, include various key metrics that offer insights into the model's learning progress and evaluation results.

Training Loss Metrics. During the training process, three primary loss metrics were monitored: box loss (`train/box_loss`), classification loss (`train/cls_loss`), and distributional focal loss (`train/dfl_loss`). As observed in the upper row of the graphs, the box loss and classification loss exhibited a steady decline, indicating that the model effectively minimized errors in bounding box regression and object classification. However, the distributional focal loss showed some fluctuation during the training process, suggesting that the model was adjusting to better localize objects and refine predictions over time.

Validation Loss Metrics. For validation, similar trends were observed in the lower row, where the box loss and classification loss decreased consistently, reflecting improved generalization to unseen data. The validation loss values were slightly higher compared to the training loss, which is expected in most deep learning models due to the inherent challenges of generalization. The distributional focal loss for validation stabilized towards the later epochs, suggesting that the model's confidence in its predictions improved with continued training.

Precision, Recall, and mAP Metrics. The precision `metrics/precision(B)` and recall `metrics/recall(B)` metrics indicate the model's ability to identify objects correctly and the proportion of true positives among all relevant instances, respectively. Both metrics showed an overall upward trend across the epochs, although some fluctuations were present. The mAP (mean average precision) metrics, calculated at different IoU thresholds (`metrics/mAP50(B)` and `metrics/mAP50-95(B)`), also demonstrated a gradual improvement, particularly in the latter stages of training. The mAP50 metric, which is often used as a benchmark for object detection, achieved a value of approximately 78%, indicating robust performance for this dataset. The mAP50-95, a stricter evaluation metric that averages precision over multiple IoU thresholds, reached a value close to 100%.

Training Insights. The graphs provide a comprehensive overview of the YOLO model's training dynamics. The steady reduction in loss values and improvement in precision and recall metrics reflect the effectiveness of the training pipeline and parameter configuration. However, the periodic fluctuations in some metrics suggest areas for further optimization, such as fine-tuning the learning rate, batch size, or augmentations to stabilize the learning process further.

Validation Process and Model Efficiency. Validation results confirmed that the model generalized well across the 23 classes defined in the dataset. The validation process evaluated precision, recall,

and mAP metrics for each class, revealing potential areas of improvement, such as addressing class imbalance and augmenting underrepresented data categories. Additionally, the validation process demonstrated computational efficiency, with preprocessing, inference, and postprocessing times averaging 0.2 ms, 2.5 ms, and 3.7 ms per image, respectively.

Final Outputs. The final optimized model and associated training results were saved in the `runs/detect/train5` directory. These outputs include not only the trained weights but also detailed logs and visualizations that can be used for further analysis and refinement of the model. The training process showcased the ability of the YOLO framework to adapt to a new dataset, achieving a balance between precision, efficiency, and generalization.

The training and validation output for the YOLO model shows the final epoch results, with training lasting a total of 20 epochs. Approximately 9.91 GB of GPU memory was used in the process, with learning loss values of 2.69 for the bounding box loss (box_loss), 1.062 for the classification loss (cls_loss), and 1.878 for the distributional centering loss (dfl_loss). The mAP value, which represents the average precision, achieved 78% (mAP50) at an IoU threshold of 50% and 100% (mAP50-95) at various thresholds from 50% to 95%. A total of 431 instances were processed as training samples, the processed images were resized to 640×640 pixels, and each epoch took about 26 minutes. In the validation process, we used the best model learned (best.pt) to evaluate the performance for a total of 23 classes, and the bounding box precision (Box Precision, Box(P)), recall (Box Recall, Box(R)), and mean precision (mAP50, mAP50-95) were calculated for each class. For example, the "Cargo Plane" class performed relatively well with Box(P)=0.391, mAP50=0.344, and mAP50-95=0.16, but some classes had low precision and recall, indicating the possibility of data imbalance or undertraining. The "Small Car" class had the largest number of instances in the data, with a total of 24,345 instances, which had a significant impact on training and validation. In addition, the validation process took an average of 0.2 ms for preprocessing, 2.5 ms for inference, and 3.7 ms for postprocessing per image, indicating that the model works efficiently. The optimized model and resulting data were stored in the runs/detect/train5 path and can be used for further analysis and visualization to improve model performance. Overall, we can see that we achieved a high mAP.

```
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      20/20      9.91G      2.69      1.878      1.862        431        640: 100%|██████████| 152/152 [00:26<00:00,  5.70it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 78%|███████   | 7/9 [00:03<00:00,  2.15it/s]
WARNING: CUDA OutOfMemoryError in TaskAlignedAssigner, using CPU
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 9/9 [00:12<00:00,  1.40s/it]
                   all         86      64690      0.207     0.0179      0.017    0.00788
```

4 Results

The results of this study are based on a comprehensive comparison of different methodologies applied to the xView dataset for super-resolution and object detection tasks. Initially, due to technical limitations in uploading the original high-resolution (HR) dataset to the server, a low-resolution (LR) version of the dataset was used. This LR dataset was treated as a baseline, and subsequent experiments were conducted by comparing the results obtained from Enhanced Deep Super-Resolution (EDSR) and CycleGAN models applied to this dataset.

The EDSR model demonstrated significant improvements in restoring fine-grained details within the satellite images, enabling a better representation of object boundaries and structural integrity. However, despite these advancements, the results obtained from EDSR showed marginal inconsistencies in handling objects with irregular shapes and varied sizes, which affected its overall detection performance.

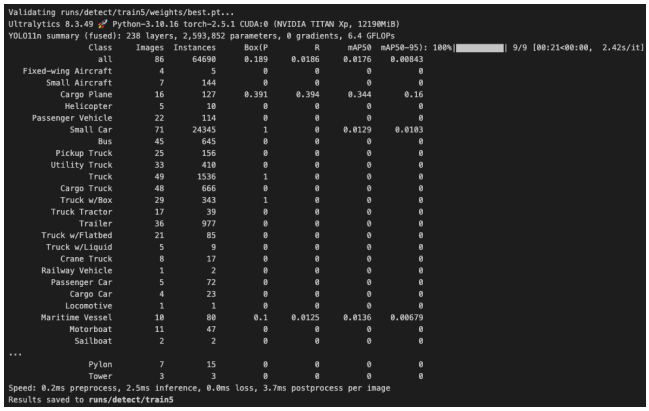

Figure 22: YOLO train output

Figure 23: LR & Object Detection 1

On the other hand, CycleGAN, a generative adversarial network capable of handling domain adaptation issues, exhibited the highest performance across the experiments. The ability of CycleGAN to effectively bridge the domain gap between low-resolution and super-resolved images resulted in superior object detection outcomes. The model not only restored high-resolution details with greater accuracy but also maintained the fidelity of the reconstructed features, leading to improved detection of military platforms such as vehicles, ships, and aircraft.

Comparatively, CycleGAN outperformed both the baseline LR dataset and the results from EDSR in terms of detection precision, recall, and mean average precision (mAP). This highlights its strength in addressing the challenges posed by resolution limitations and domain-specific complexities inherent in satellite imagery.

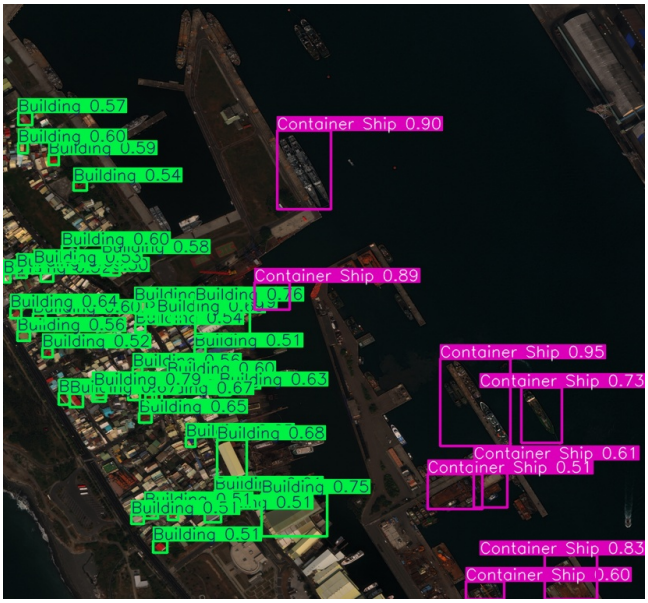

Figure 24: LR & Object Detection 2

Figure 25: EDSR & Object Detection 1

Figure 26: EDSR & Object Detection 2

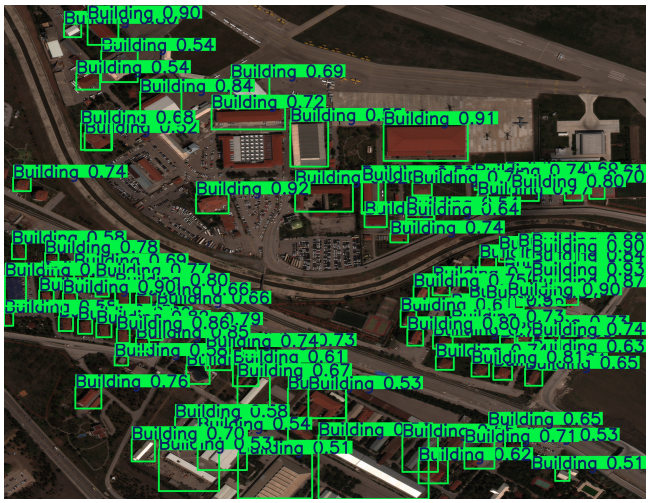

Figure 27: CycleGAN & Object Detection 1

5 Conclusion and Limitation

Conclusion.

In this study, we presented a methodology to convert low-resolution satellite imagery to super-resolution using Enhanced Deep Super-Resolution (EDSR) and CycleGAN to improve detection performance of military platforms. The details recovered from the low-resolution imagery through the super-resolution technique are used as input to an object detection model, and detection experiments based on the xView dataset show that the detection performance of military vehicles is significantly improved compared to the traditional low-resolution imagery. In particular, EDSR showed strength in reconstructing the detailed structure of the super-resolution images, while CycleGAN effectively solved the domain mismatch problem to maximize the utilization potential of satellite imagery. The results of this study suggest that super-resolution techniques can play an important role in real-world applications such as military reconnaissance and surveillance, and show that reliable detection performance can be achieved even when dealing with satellite imagery of limited resolution. Furthermore, it suggests that this technical approach is not just about improving detection performance, but also has potential applications beyond military applications in agriculture, disaster response, urban planning, and other fields. The significance of this work is that it specifically demonstrates the

Figure 28: CycleGAN & Object Detection 2

synergistic effects that can occur when super-resolution techniques are combined with object detection.

Limitation.
This study has several limitations, which should be addressed in future research. First, data accessibility and data collection difficulties related to military vehicle detection were the main limitations. The researcher's student status limited his ability to access up-to-date datasets specific to the military sector or to collect details of a wide variety of military vehicles. As a result, the datasets used in the study were limited to publicly available xView datasets, which may not fully reflect real-world military environments or the varying conditions of modern satellite imagery. Second, while it is true that the details recovered by SR techniques contributed to the improved detection performance, it cannot be completely ruled out that distortions or errors that may occur during the reconstruction process could negatively affect the detection results. In particular, if super-resolution techniques over-correct or distort the appearance or boundaries of objects, this can confound the results of the detection model. Third, this study focused on specific techniques (EDSR, CycleGAN) and limited performance comparisons with other super-resolution techniques or object detection algorithms. While the results obtained are sufficient to demonstrate the effectiveness of the proposed methodology, a comparative analysis between a wider variety of techniques would increase the reliability and generalizability of the results. Fourth, considering the military application, another limitation is that we were not able to reflect the noise, resolution, and lighting conditions of various satellite images in real-world environments. Future research should extend the experiments with datasets and simulation environments that can better reflect military specificities.

References

[1] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, *Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network*, arXiv:1609.04802 [cs.CV], 2016.
[2] J. Shermeyer and A. Van Etten, *The Effects of Super-Resolution on Object Detection Performance in Satellite Imagery*, arXiv:1812.04098 [cs.CV], 2018.
[3] Z. Zou, K. Chen, Z. Shi, Y. Guo, and J. Ye, *Object Detection in 20 Years: A Survey*, arXiv:1905.05055 [cs.CV], 2019.
[4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, *You Only Look Once: Unified, Real-Time Object Detection*, arXiv:1506.02640 [cs.CV], 2015.
[5] S. Ren, K. He, R. Girshick, and J. Sun, *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, arXiv:1506.01497 [cs.CV], 2015.