

I have attempted to write a GAN before (probably about two years ago now), but really should've waited until I was on or had access to better hardware. I vaguely remember that I had trouble getting the training rates to sync and wasn't having any success. Can you write a template program using Python that represents a starter GAN?

Answer

Certainly! I'd be happy to provide you with a template program for a starter Generative Adversarial Network (GAN) using Python and TensorFlow. This template will give you a basic structure to build upon and experiment with. Keep in mind that GANs can be complex and may require some tweaking to achieve good results, especially for specific tasks.

Here's a basic GAN template that generates simple 2D data:

```
python □
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Set random seed for reproducibility
tf.random.set_seed(42)
np.random.seed(42)

# Define the Generator model
def make_generator_model():
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(16, input_shape=(100,), activation='relu'),
        tf.keras.layers.Dense(32, activation='relu'),
        tf.keras.layers.Dense(2) # Output 2D data points
    ])
    return model

# Define the Discriminator model
def make_discriminator_model():
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(32, input_shape=(2,), activation='relu'),
        tf.keras.layers.Dense(16, activation='relu'),
        tf.keras.layers.Dense(1, activation='sigmoid')
    ])
    return model

# Create the generator and discriminator
generator = make_generator_model()
discriminator = make_discriminator_model()

# Define loss functions
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=False)

# Define optimizers
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# Training step
@tf.function
def train_step(real_samples, batch_size):
    noise = tf.random.normal([batch_size, 100])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
```

```

generated_samples = generator(noise, training=True)

real_output = discriminator(real_samples, training=True)
fake_output = discriminator(generated_samples, training=True)

gen_loss = cross_entropy(tf.ones_like(fake_output), fake_output)
disc_loss = cross_entropy(tf.ones_like(real_output), real_output) + \
            cross_entropy(tf.zeros_like(fake_output), fake_output)

gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))

return gen_loss, disc_loss

# Training loop
def train(dataset, epochs, batch_size):
    for epoch in range(epochs):
        for batch in dataset:
            gen_loss, disc_loss = train_step(batch, batch_size)

            if (epoch + 1) % 10 == 0:
                print(f"Epoch {epoch+1}, Gen Loss: {gen_loss:.4f}, Disc Loss: {disc_loss:.4f}")

            # Generate and plot samples every 50 epochs
            if (epoch + 1) % 50 == 0:
                generate_and_plot_samples(generator, epoch + 1)

# Generate and plot samples
def generate_and_plot_samples(generator, epoch):
    noise = tf.random.normal([100, 100])
    generated_samples = generator(noise, training=False)

    plt.scatter(generated_samples[:, 0], generated_samples[:, 1])
    plt.title(f"Generated Samples at Epoch {epoch}")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.savefig(f"generated_samples_epoch_{epoch}.png")
    plt.close()

# Generate a simple dataset (a circle)
def generate_real_samples(n):
    radius = 5
    theta = np.linspace(0, 2*np.pi, n)
    x = radius * np.cos(theta) + np.random.normal(0, 0.5, n)
    y = radius * np.sin(theta) + np.random.normal(0, 0.5, n)
    return np.column_stack((x, y))

# Prepare the dataset
BATCH_SIZE = 32
BUFFER_SIZE = 1000
EPOCHS = 500

real_data = generate_real_samples(BUFFER_SIZE)
dataset = tf.data.Dataset.from_tensor_slices(real_data).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

```

```
# Train the GAN
train(dataset, EPOCHS, BATCH_SIZE)
```

This template includes:

1. A simple Generator that takes random noise and produces 2D data points.
2. A Discriminator that tries to distinguish between real and generated data.
3. Loss functions and optimizers for both networks.
4. A training loop that alternates between training the Generator and Discriminator.
5. A function to generate and plot samples during training.
6. A simple dataset generator that creates points in a circular pattern.

To use this template:

1. Make sure you have TensorFlow, NumPy, and Matplotlib installed.
2. Run the script, and it will train the GAN for 500 epochs, printing loss values every 10 epochs and generating sample plots every 50 epochs.

Remember that this is a basic template, and you may need to adjust the architecture, hyperparameters, and training process depending on your specific use case and the complexity of the data you're working with.

Some tips for improving GAN training:

1. Experiment with different learning rates for the generator and discriminator.
2. Try different architectures for both networks.
3. Implement techniques like label smoothing or adding noise to the discriminator inputs.
4. Use more advanced GAN variants like WGAN or DCGAN for better stability.

Good luck with your GAN experiments!

 Share  Rewrite



I'm not familiar with using raw Tensorflow (have a bit more experience with Keras). In the above code, where is the real world data coming in?

Sources

[Tensorflow 2.0: Best way for structure the output of `tf.data.Dataset`](#)

[Why should one use pure TensorFlow over Keras? - Stack Overflow](#)

...

 stackoverflow

 stackoverflow

 tensorflow

Answer

In the provided GAN template, the real-world data is generated synthetically using the `generate_real_samples` function. This function creates a simple circular pattern of 2D points with some added noise ^{5①}. Specifically, this part of the code handles the real data generation:

python

```
def generate_real_samples(n):
    radius = 5
    theta = np.linspace(0, 2*np.pi, n)
    x = radius * np.cos(theta) + np.random.normal(0, 0.5, n)
    y = radius * np.sin(theta) + np.random.normal(0, 0.5, n)
    return np.column_stack((x, y))
```



```
# Prepare the dataset
BUFFER_SIZE = 1000
real_data = generate_real_samples(BUFFER_SIZE)
dataset = tf.data.Dataset.from_tensor_slices(real_data).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

This synthetic data is then used to create a TensorFlow Dataset object, which is shuffled and batched for training ³. In a real-world scenario, you would replace this synthetic data generation with actual data loading, such as reading images from files or loading data from a database ¹ ³.

[Share](#) [Rewrite](#)

...



[Watch](#)

If I wanted to use a GAN to make a discriminator learn the base patterns behind stars in the sky in Stellarium (after putting the Ocular plugin on in an attempt to try to make it as close as possible to a real world camera), would it make sense to alternate between footage from that real camera and the generator, even though the real world camera would be limited in location?

❖ Sources

[Generating and upscaling images of nebula with FastGAN! - Reddit](#)

[How to Find the Perfect Astrophotography Target with Stellarium](#)

[Generating images of the M8](#)

reddit

photographingspace

academic.oup

❖ Answer

Using a GAN to learn the patterns of stars in the sky from Stellarium and real-world camera footage is an interesting approach, but there are some considerations to keep in mind:

1. Data consistency: Alternating between Stellarium images and real camera footage could potentially work, but it may introduce inconsistencies in the data ¹. The Stellarium images, even with the Ocular plugin, might not perfectly match real-world camera characteristics.

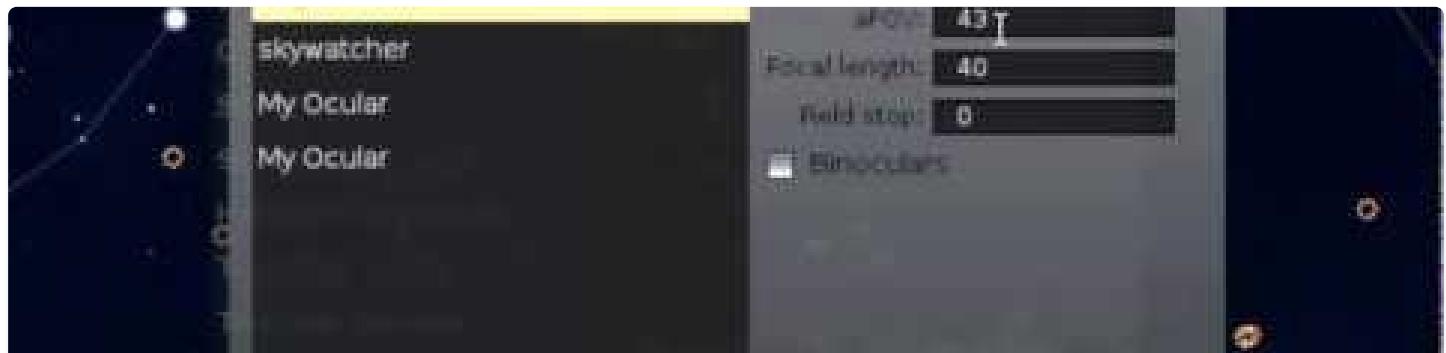
2. Limited real-world data: As you mentioned, the real-world camera would be limited in location, which could bias the discriminator towards patterns specific to that location ³. This might not generalize well to other parts of the sky.
3. Image quality differences: Stellarium generates idealized images, while real camera footage may include noise, atmospheric effects, and optical aberrations ². This discrepancy could confuse the discriminator.

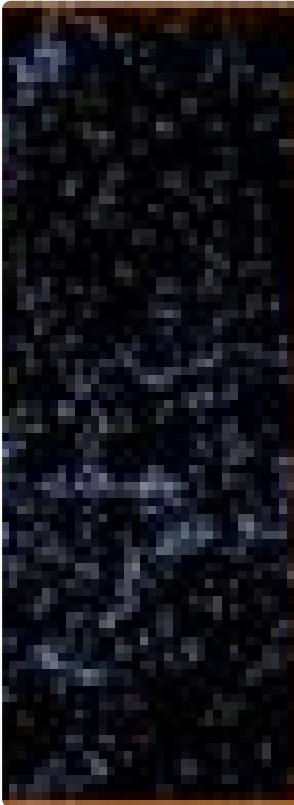
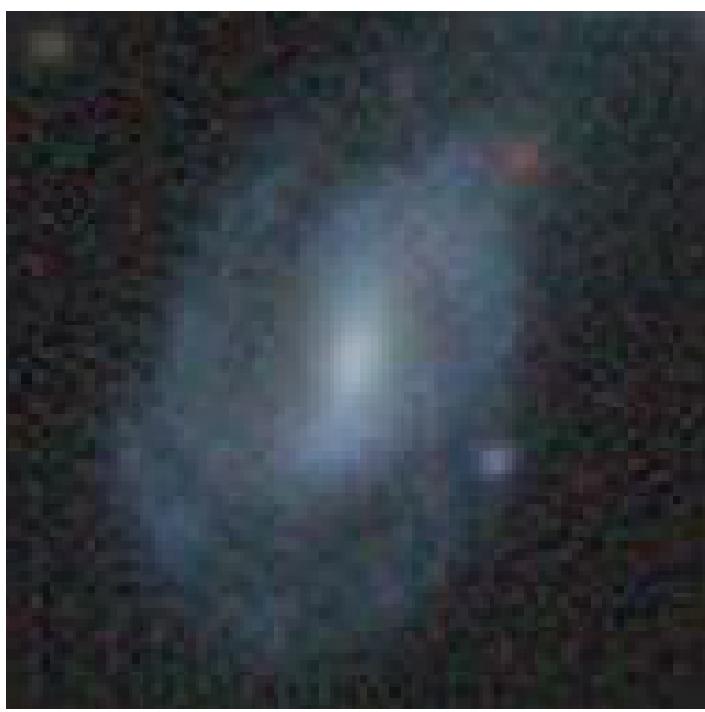
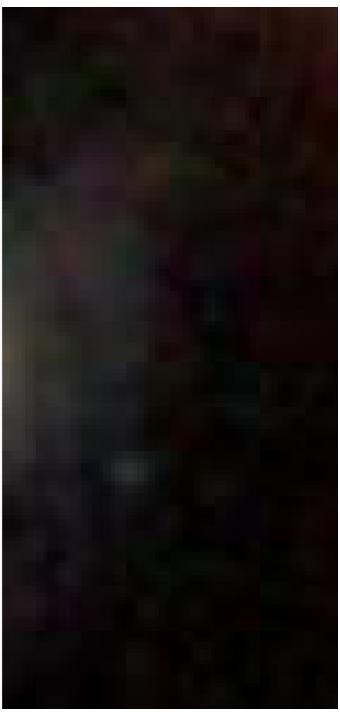
To address these issues, you could:

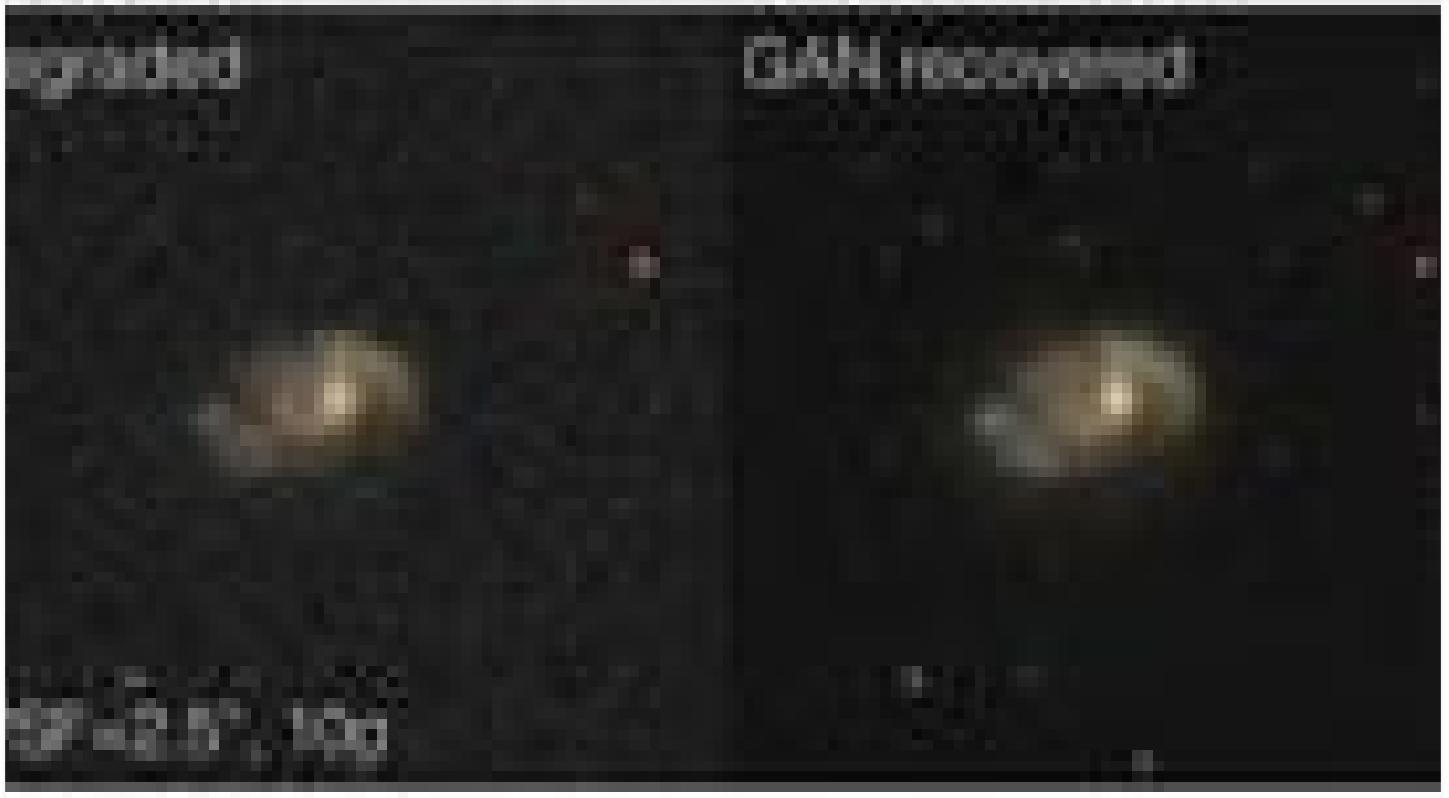
1. Use Stellarium to generate a diverse set of sky images from various locations and times, simulating different atmospheric conditions and camera settings ⁴.
2. Apply realistic noise and distortions to the Stellarium images to make them more similar to real camera footage ¹.
3. Augment your limited real-world data with techniques like rotation, flipping, and small translations to increase variety ³.
4. Consider using a conditional GAN (cGAN) approach, where you provide additional information about the image source (Stellarium or real camera) to help the model learn the differences ³.
5. Implement a progressive growing technique, starting with low-resolution images and gradually increasing complexity, which could help in learning stable patterns ³.

By carefully preprocessing your data and designing your GAN architecture, you could potentially create a model that learns generalizable patterns of stars in the sky, combining the benefits of both Stellarium's comprehensive sky simulation and real-world camera characteristics.

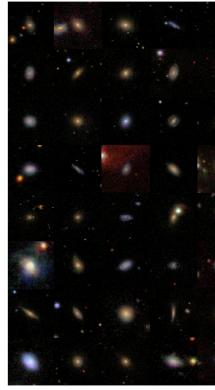
Share Rewrite







generated images from trained on scraped im-



(b) Examples of generated images from Lightweight GAN trained on scraped im-

Dataset.

View More

If I'm just trying to teach a discriminator network the names of 5000-6000 stars, planets, moon, and any other commonly visible objects in the sky so it can label them, what network architecture for that segment of the GAN should I make? I think I'd like it to put squares around objects it thinks it recognizes, like a YOLO, maybe generate a false color image with recognized object pixels in green and unsure in red.

• Sources

Answer

To create a discriminator network for recognizing and labeling celestial objects, you can adapt a YOLO (You Only Look Once) architecture, which is well-suited for object detection tasks. Here's a suggested architecture for your discriminator:

1. Input layer: Accept images of the night sky (e.g., 416x416 or 608x608 pixels, with 3 color channels).
2. Backbone: Use a pre-trained convolutional neural network (CNN) like ResNet or DarkNet as the feature extractor. This will help in learning robust features from the input images ¹ ₄.
3. Neck: Implement a feature pyramid network (FPN) or LASF_Neck for multi-scale feature fusion, which is crucial for detecting objects of various sizes (from small stars to larger planets or moons) ² ₂.
4. Detection head: Use multiple detection heads at different scales, similar to YOLOv3 or YOLOv5, to predict bounding boxes, object confidence, and class probabilities ¹ ₃.
5. Output: The network should output bounding box coordinates, confidence scores, and class probabilities for each detected object.

To implement the color-coding feature:

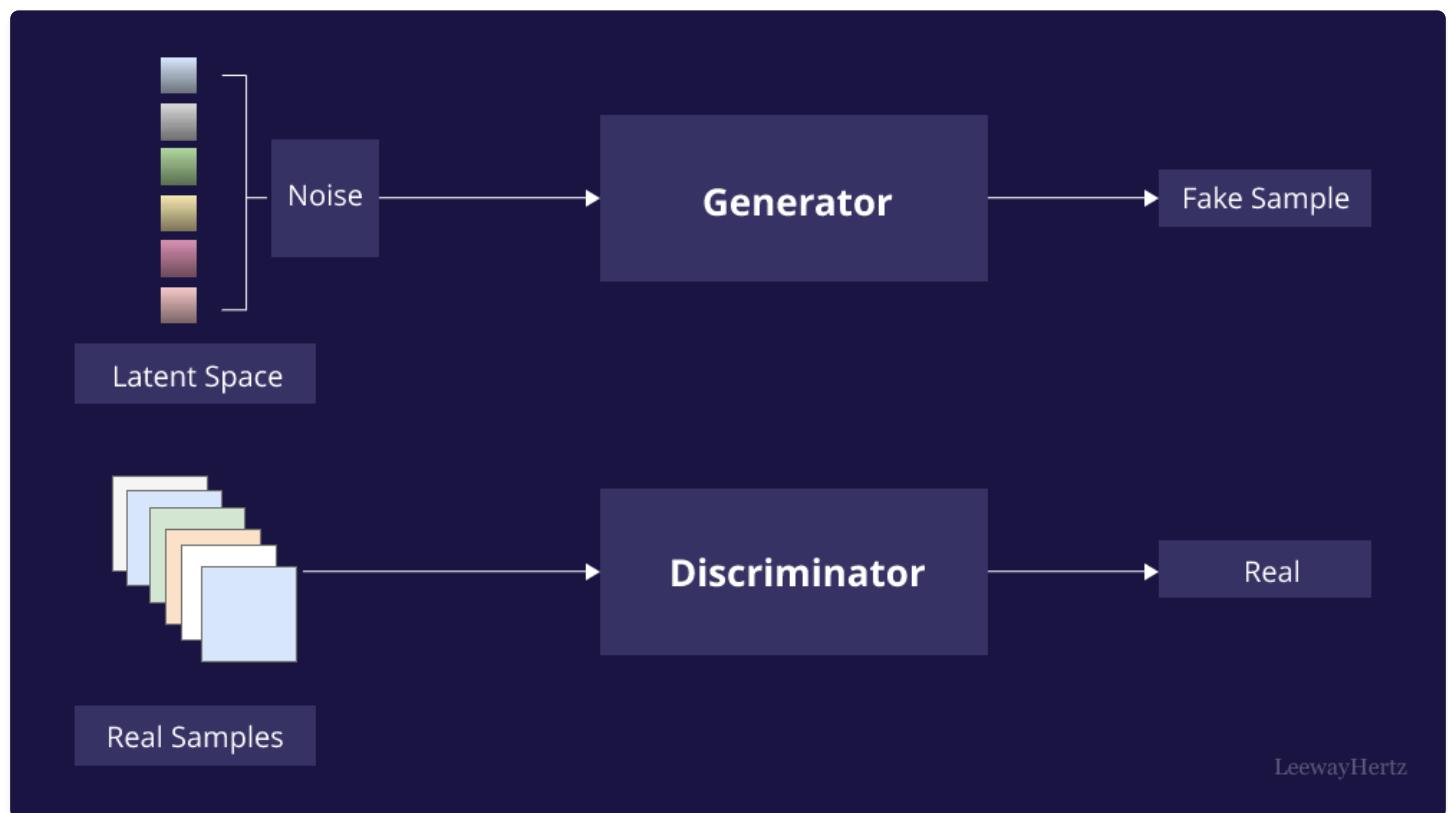
1. Post-processing step: After the network makes predictions, apply a threshold to the confidence scores.
2. For objects above the threshold, draw green bounding boxes and label them.
3. For objects below the threshold but above a lower cutoff, color the pixels red to indicate uncertainty.

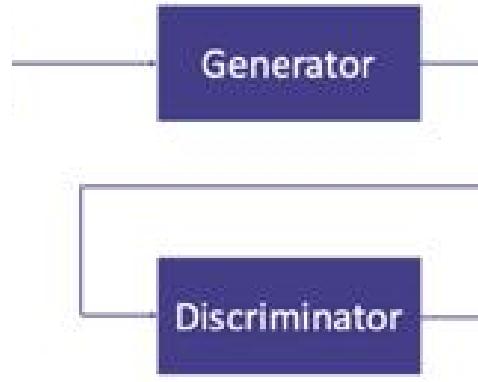
This architecture combines elements from YOLO-Dynamic and SOD-YOLO, which are specifically designed for space object detection ² ₃. The multi-scale approach will help in detecting both small stars and larger celestial bodies, while the attention mechanisms (like SimAM) can improve the network's ability to focus on relevant features in the complex night sky images ³ ₃.

Remember to train this network with a diverse dataset of night sky images, including various lighting conditions and object positions. You may also need to fine-tune the network architecture based on your specific requirements and the characteristics of your dataset.

 Share  Rewrite





View More

Seems like I may be somewhat complicating my problem space still. For instance, objects at infinity mostly move at the same speed with Earth's rotation and if I have two camera's focused for parallax, anything that can be located by SLAM to be closer would immediately define a patch of pixels that needs to be classified or marked as potentially anomalous. For the objects at infinity (or much further like planets), they'd still go into that other box for processing.

Answer

You're absolutely right, and I appreciate you bringing this up. Your insight about using parallax and SLAM (Simultaneous Localization and Mapping) to differentiate between nearby objects and those at "infinity" is excellent. This approach can indeed simplify your problem space and make the classification task more manageable. Let's break down this revised approach:

1. Dual Camera Setup with SLAM:
 - Use two cameras set up for stereo vision.
 - Implement SLAM to identify objects with measurable parallax.
2. Object Separation:
 - Near objects (with parallax): These would be things like satellites, meteors, or potentially unidentified flying objects.
 - "Infinity" objects: Stars, planets, and other celestial bodies.
3. Classification Pipeline:
 - a. For near objects:
 - These objects will move differently relative to the star field.
 - Use a simpler object detection and tracking algorithm.
 - Classify these as "anomalous" or into specific categories (e.g., "satellite", "meteor").
 - b. For "infinity" objects:
 - These will move uniformly with Earth's rotation.
 - Use a star pattern recognition algorithm (like plate solving in astrophotography).
 - Match against a database of known stars, planets, and deep-sky objects.

4. Neural Network Architecture:

Instead of a complex YOLO-like architecture, you could use:

- A CNN for feature extraction
- A transformer or attention mechanism to handle the spatial relationships between stars
- Output layer that predicts object class and confidence for each region of interest

5. Post-processing:

- For identified objects: Draw green bounding boxes and labels.
- For unidentified or low-confidence objects: Highlight in red.

This approach leverages the known physics of celestial object movement and your dual-camera setup, potentially reducing the complexity of your neural network and improving accuracy. It also naturally separates the task of identifying anomalous near-Earth objects from the task of recognizing known celestial bodies.

 Share  Rewrite



Is per pixel classification something that is necessarily done by a ton of multi-heads with different sized boxes using unions between them and the like?

⦿ Sources

[Difference between Object Detection and Pixel Classification](#)

[Distinction between pixel-based and object based classification?](#)

[Image classification vs detection](#)

 geowgs84

 gis.stackexchange

 docs.clarifai

✳ Answer

Per pixel classification is not necessarily done using multi-heads with different sized boxes and unions between them. This approach is more commonly associated with object detection techniques like YOLO (You Only Look Once) ¹.

Pixel classification, also known as semantic segmentation, typically assigns a class or category to each individual pixel in an image based on its spectral, spatial, or textural characteristics ¹ ². The process usually involves:

1. Feature extraction: Using convolutional neural networks (CNNs) to extract relevant features from the image.
2. Pixel-wise classification: Applying classifiers to categorize each pixel based on the extracted features.
3. Output generation: Producing a labeled image or raster where every pixel is assigned to a specific category ¹.

Common methods for pixel classification include:

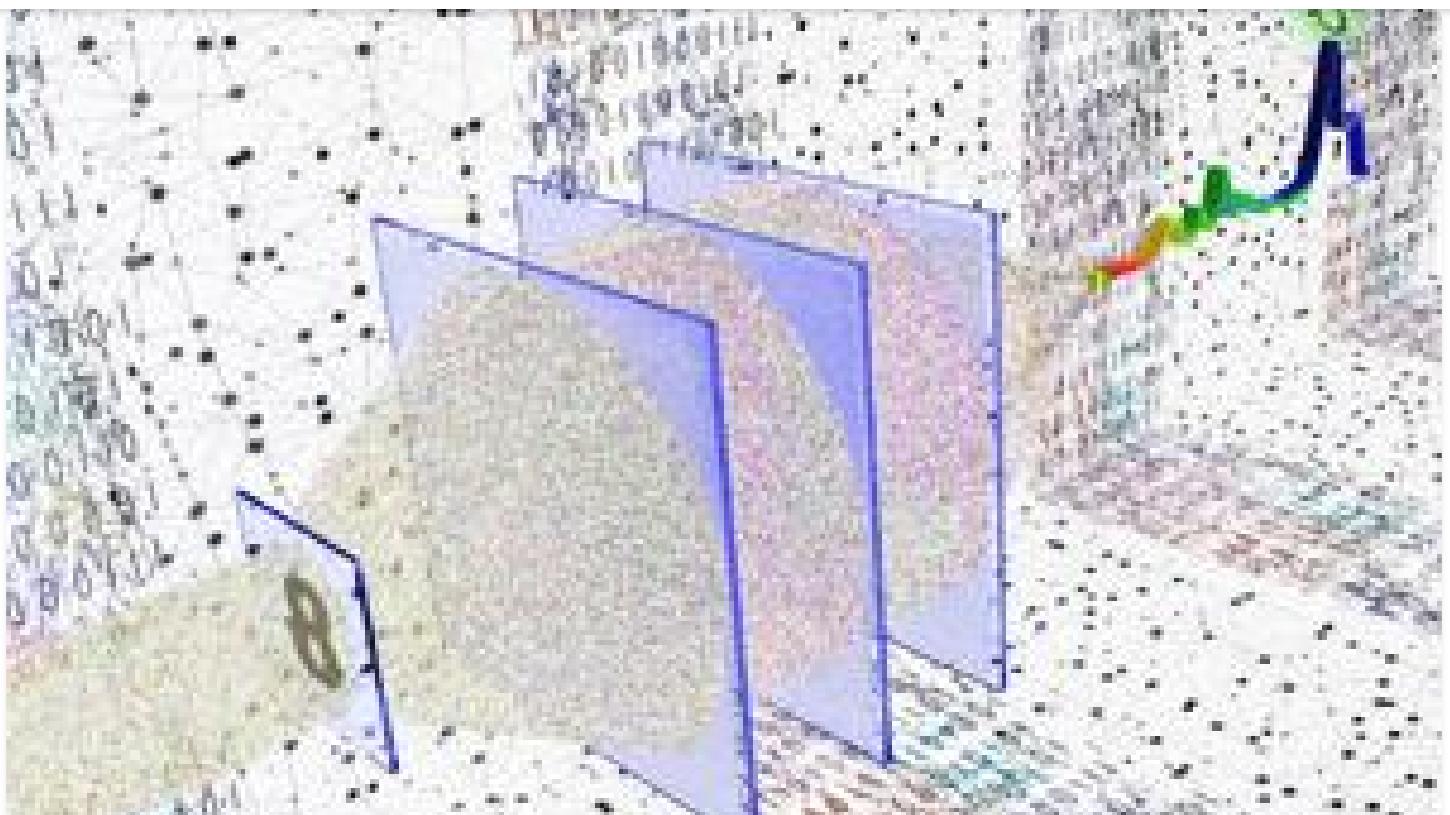
1. Traditional approaches: Maximum Likelihood Classification (MLC), Support Vector Machines (SVM), or Decision Trees.
2. Advanced techniques: Deep learning methods such as Fully Convolutional Networks (FCNs) for semantic segmentation ¹.

These methods typically process the entire image at once, assigning labels to all pixels simultaneously, rather than using multiple detection heads with varying box sizes ² ³.

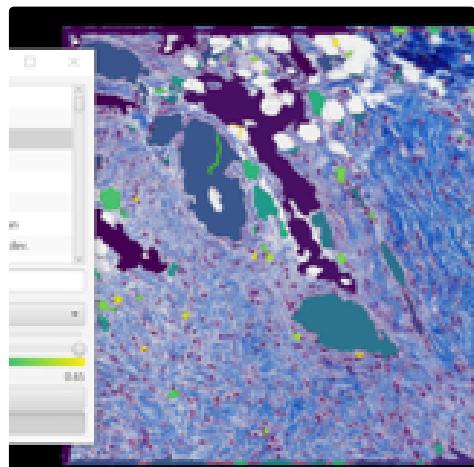
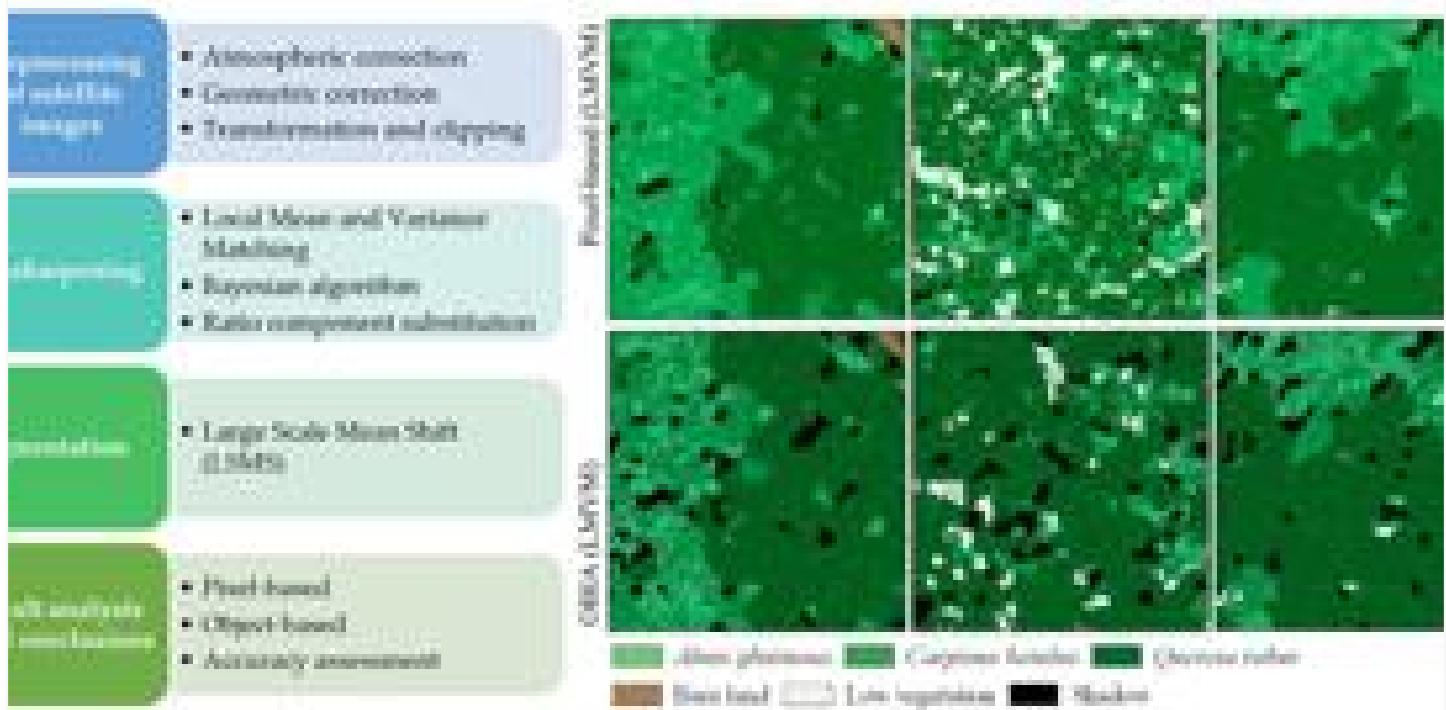
It's important to note that pixel classification differs from object detection. While object detection identifies and locates specific objects using bounding boxes, pixel classification categorizes each pixel without necessarily distinguishing between individual object instances ³ ⁴.

 Share  Rewrite





Assessment of Pixel- and Object-Based Tree Species Classification in Mixed Deciduous Forest



View More

If you were to really be confident in your pixel classification results, wouldn't you want to know that you're definitely classifying individual instances?

Answer

You raise an excellent point. To achieve high confidence in pixel classification results, especially in complex scenarios like astronomical imaging, it's indeed valuable to consider individual instances. This approach bridges the gap between semantic segmentation (pixel-wise classification) and instance segmentation (identifying individual objects). Here's how this could be approached:

1. Instance-aware Semantic Segmentation:

- Use a network architecture that combines pixel-wise classification with instance detection.
- Examples include Mask R-CNN or YOLACT (You Only Look At Coefficients).

2. Two-stage approach:

- First, perform pixel-wise classification to identify regions of interest.
- Then, apply an instance detection algorithm to these regions to separate individual objects.

3. Attention Mechanisms:

- Implement attention layers that can focus on specific regions, helping to distinguish between nearby objects.

4. Post-processing:

- Use clustering algorithms (e.g., DBSCAN) on classified pixels to group them into distinct instances.
- Apply morphological operations to refine object boundaries.

5. Confidence Metrics:

- Implement per-instance confidence scores.
- Use ensemble methods to cross-validate classifications from multiple models.

6. Contextual Information:

- Incorporate spatial relationships and known patterns of celestial objects to improve instance identification.

By combining these techniques, you can achieve both pixel-wise classification and instance awareness, leading to more confident and accurate results in your astronomical object identification task.

 Share  Rewrite



Has there ever been instance-aware semantic segmentation done with visual transformers, especially recently?

❖ Sources

[PDF] INSTANCE-AWARE 3D SEMANTIC SEGMENTATION -
OpenReview



Instance-Aware Generalized Referring Expression Segmentation



[PDF] SOIT: Segmenting Obj - AAAI



❖ Answer

Yes, instance-aware semantic segmentation has been done with visual transformers, with several recent advancements in this area:

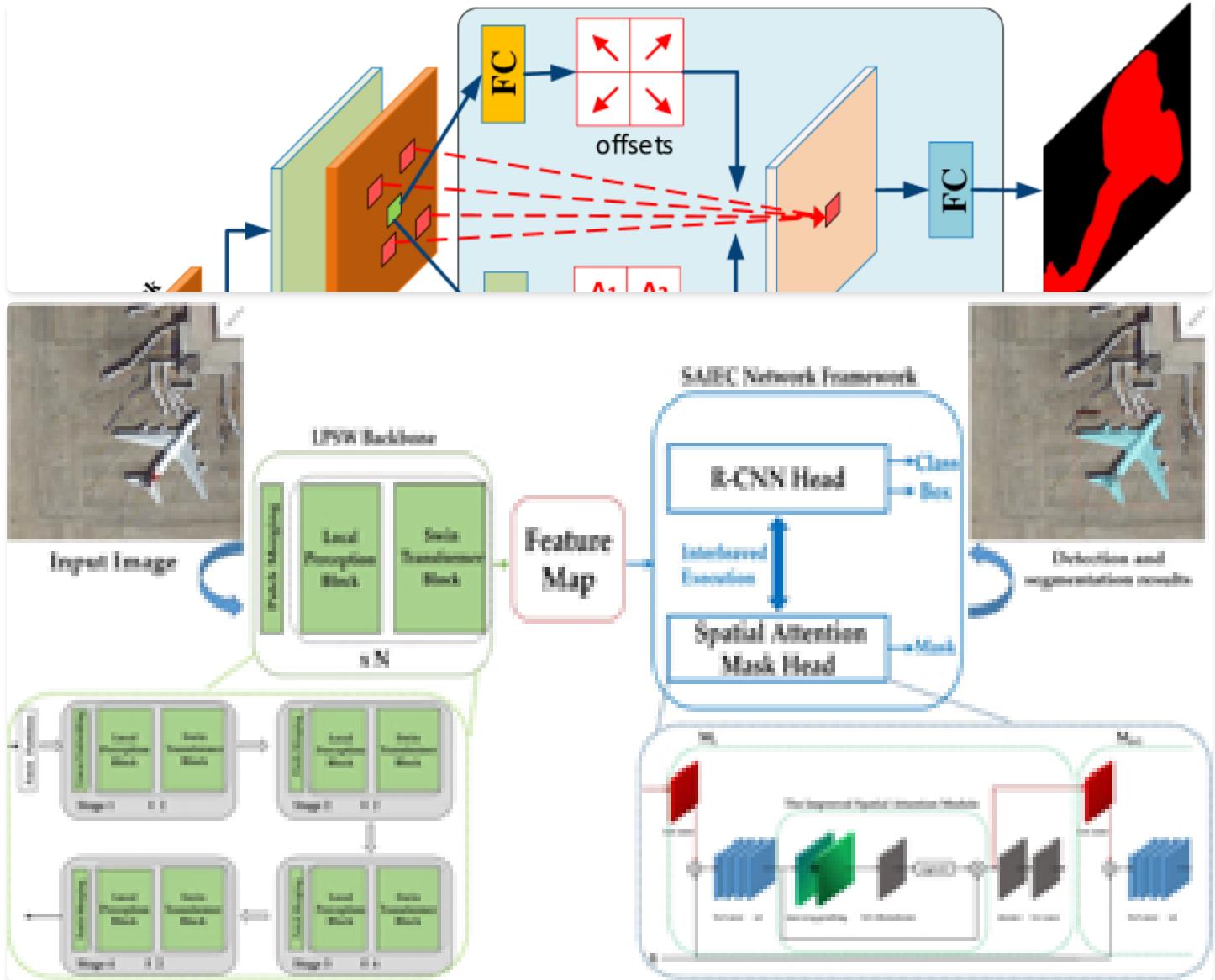
1. SOIT (Segmenting Objects with Instance-aware Transformers) is an end-to-end instance segmentation framework that uses instance-aware transformers ³ ⁵. It reformulates instance segmentation as a direct set prediction problem and eliminates the need for region proposals and non-maximum suppression.
2. InstAlign, introduced in 2023, is a novel approach for Generalized Referring Expression Segmentation (GRES) that incorporates instance-level reasoning into the segmentation process ². It uses object-level tokens to capture semantic information from both text and image inputs.
3. A 2023 paper proposed an instance-aware approach for 3D semantic segmentation that combines shape generators and classifiers to perform shape reconstruction and classification tasks for each shape instance ¹. This method significantly

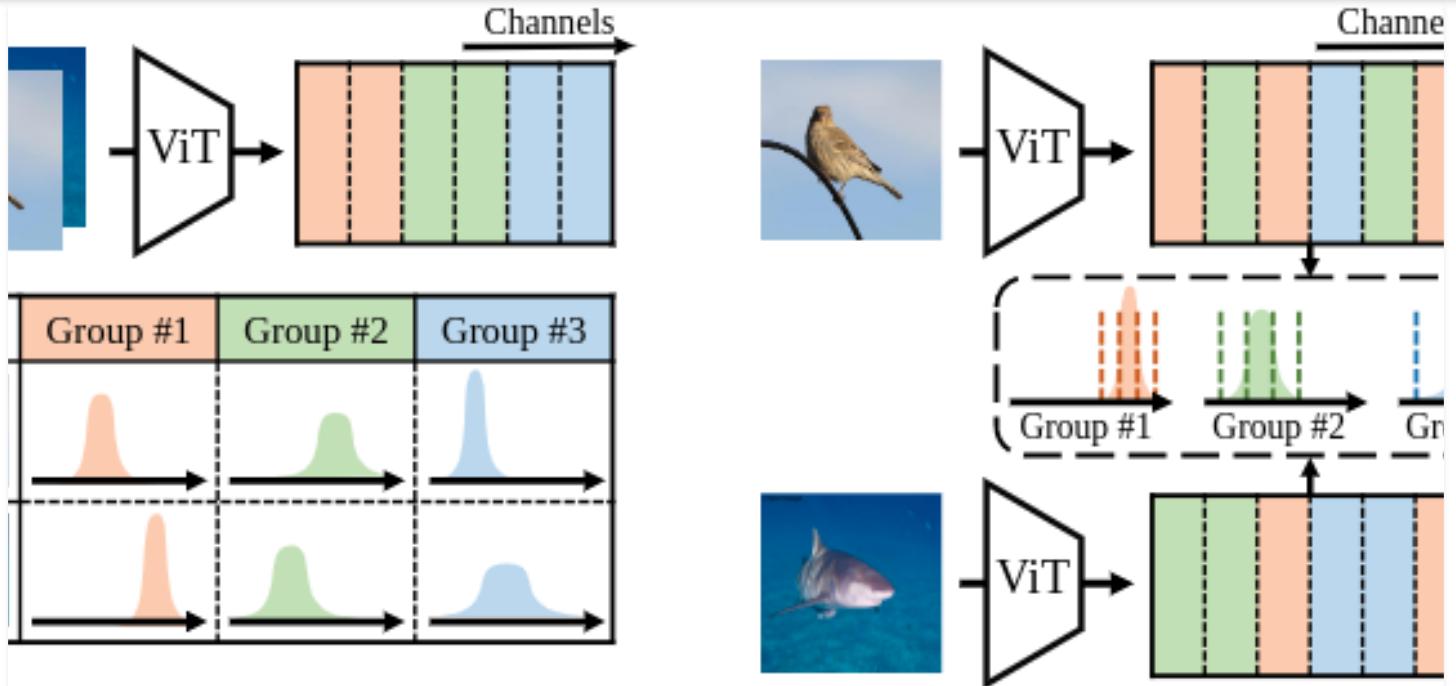
outperformed existing approaches on several public benchmarks.

4. Instance-Aware Group Quantization for Vision Transformers (IGQ-ViT) was introduced to address the challenges of post-training quantization for Vision Transformers in tasks including instance segmentation⁴. This method dynamically splits activation map channels into groups for each input instance.

These developments demonstrate that instance-aware semantic segmentation using visual transformers is an active area of research, with significant progress made in recent years across various applications including 2D and 3D segmentation tasks.

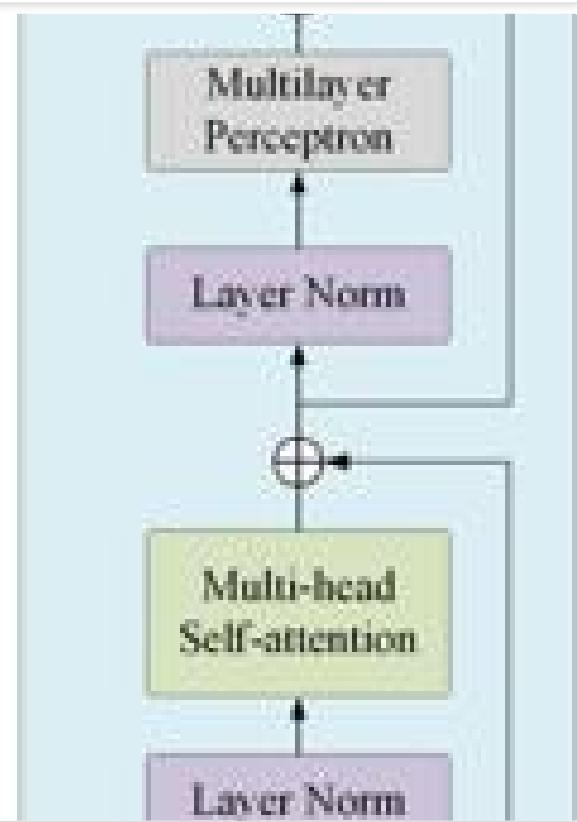
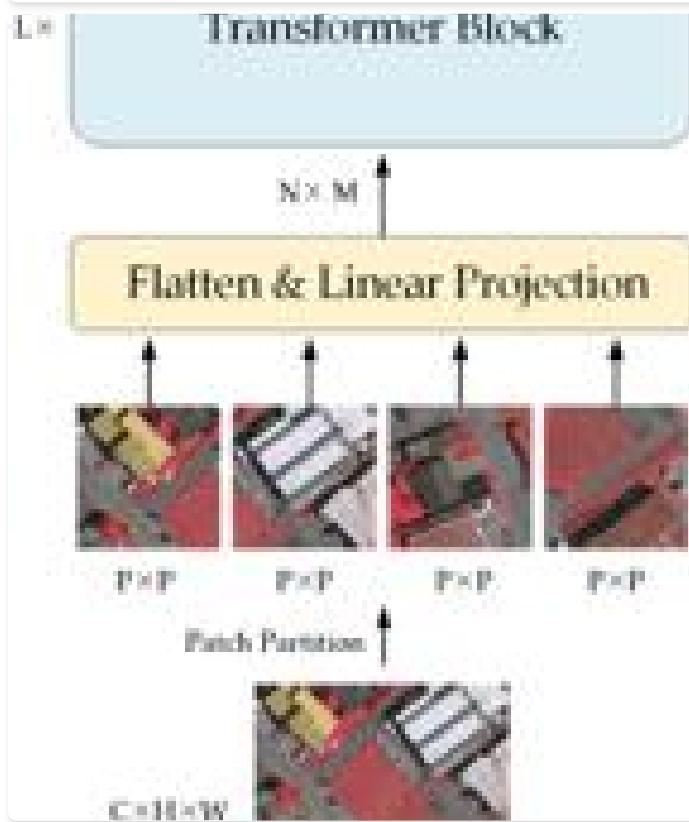
Share Rewrite

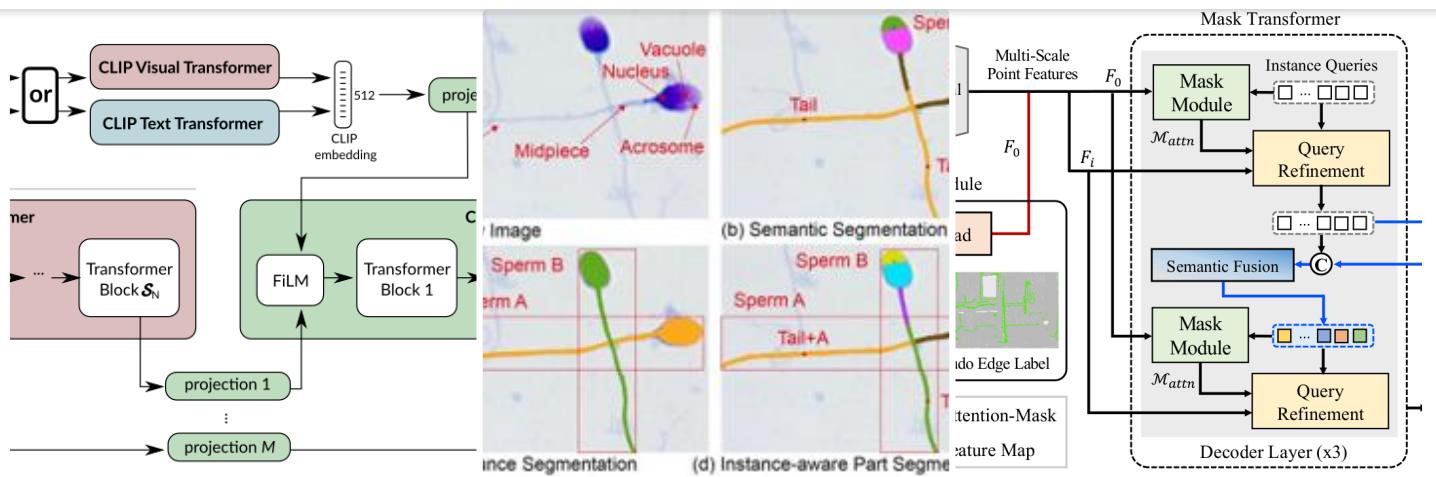




a) Group quantization

(b) IGQ-ViT





View More

OK. I think my first draft attempt will try to use a visual transformer approach. I'll forget about the GAN approach for now since Stellarium's images aren't exactly a GAN generator. For these output images, I'll need to make some code that will automatically generate bounding boxes around stars, planets, and the moon...potentially figure something out to go to the pixel level instead of a bounding box if I can. That would consist of somehow scanning through the image asking Stellarium about the contents of any pixel which it can label.

❖ Sources

[2408.13634] Enhanced Astronomical Source Classification with ...

A hybrid CNN-Transformer deep-learning object-detection model ...

Vision foundation models: ca

arxiv

academic.oup

arxiv

❖ Answer

Your approach to use a visual transformer for stellar classification and object detection is well-aligned with recent advancements in the field. Vision Transformers (ViTs) have shown promising results in astronomical image analysis, offering advantages in capturing global dependencies and contextual information ¹–³.

For generating bounding boxes and pixel-level segmentation, consider the following steps:

1. Data Generation:

- Use Stellarium's API to generate images of the night sky.
- Utilize Stellarium's object information to create labeled datasets, including pixel-level information for stars, planets, and the moon.

2. Model Architecture:

- Implement a hybrid CNN-Transformer model like AstroYOLO ². This approach combines the local feature representation strengths of CNNs with the global context capture of Transformers.
- Consider using an instance-aware semantic segmentation approach, which can provide both bounding boxes and pixel-level classifications ⁴.

3. Training Process:

- Pre-train your model on a large dataset of astronomical images.
- Fine-tune on your Stellarium-generated dataset for specific object detection and segmentation tasks.

4. Post-processing:

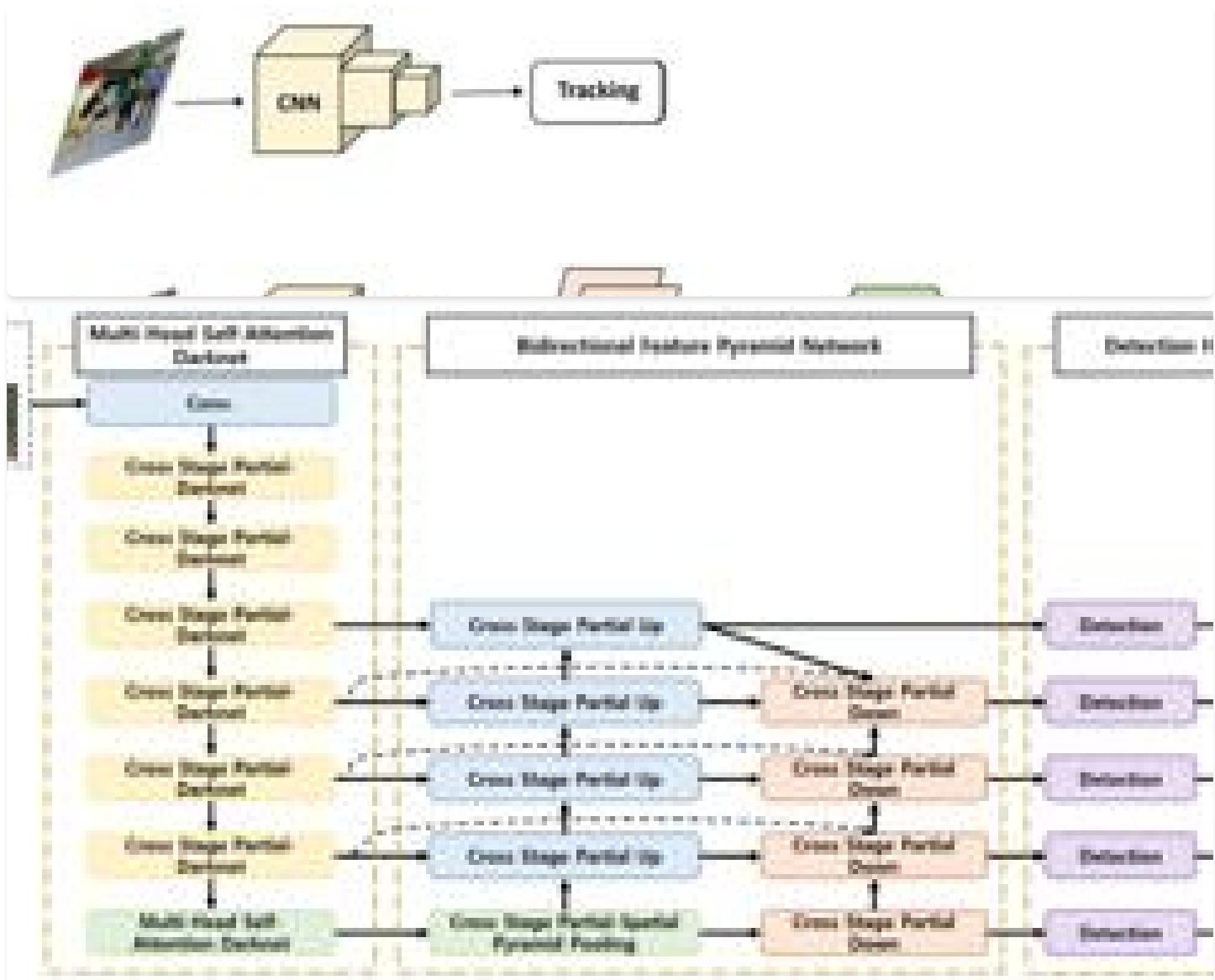
- Implement algorithms to refine the bounding boxes and segmentation masks produced by the model.
- Use Stellarium's data to validate and potentially correct the model's outputs.

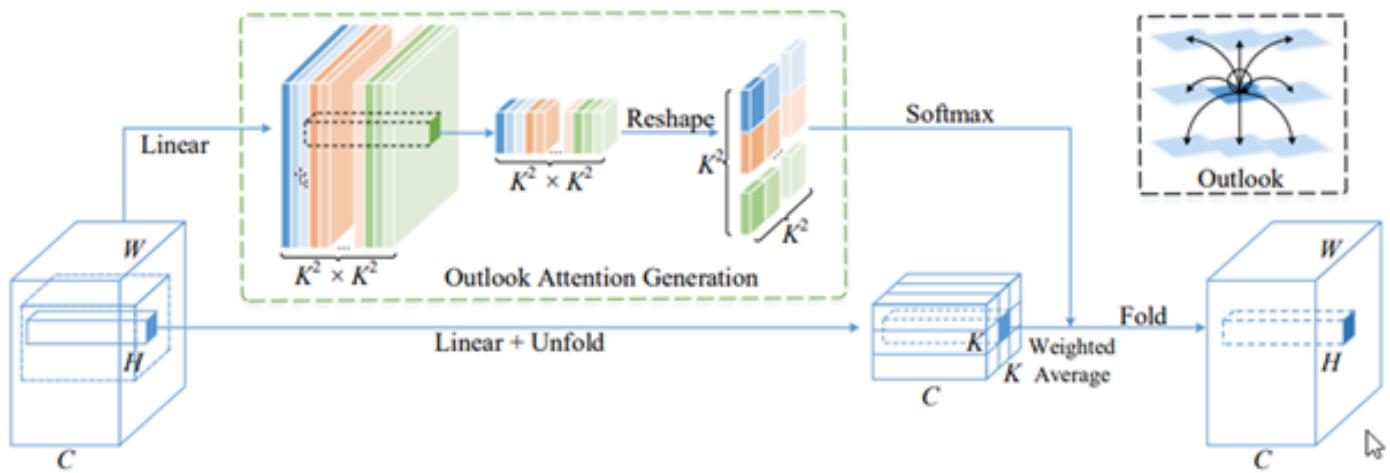
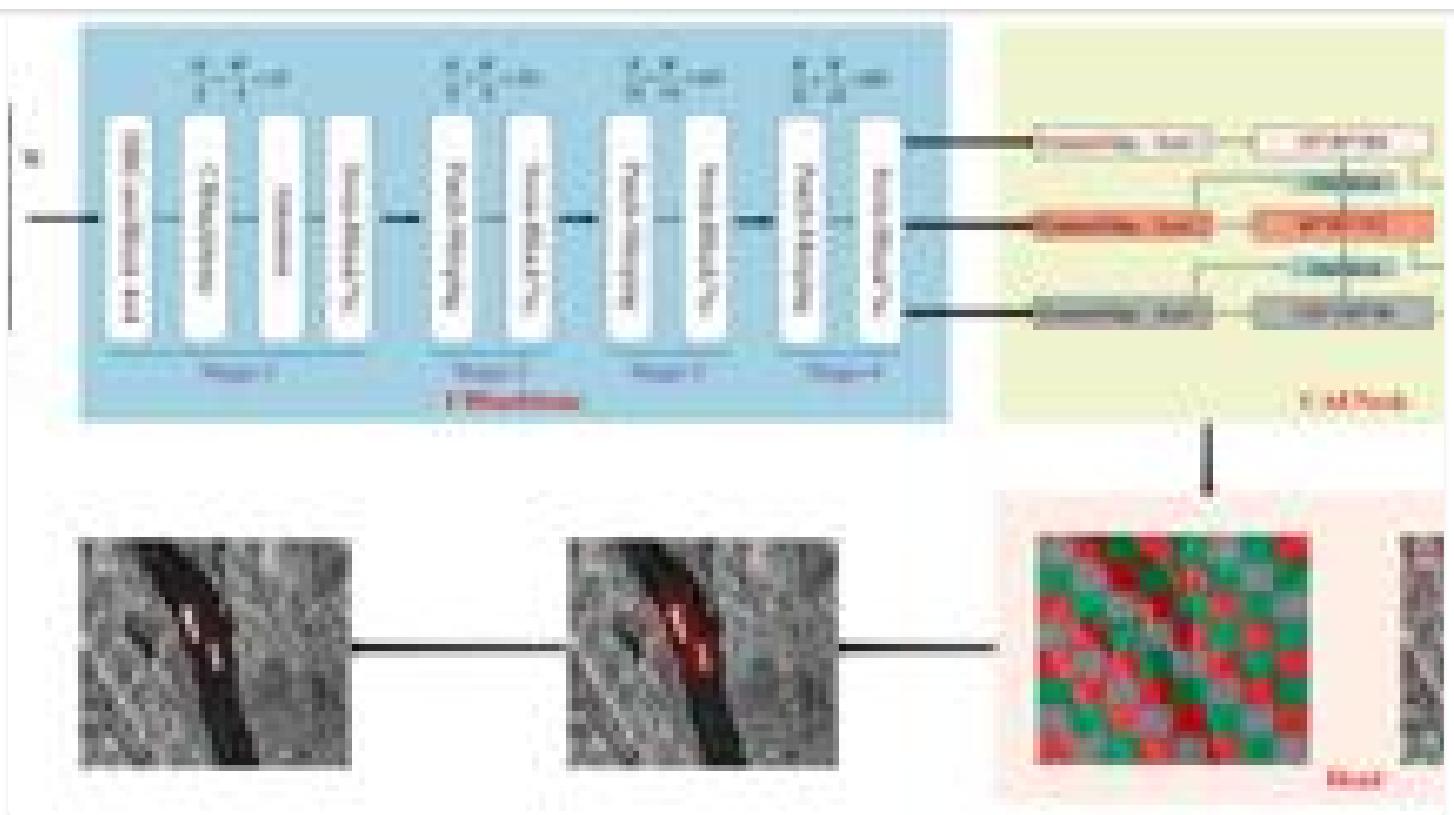
5. Pixel-level Classification:

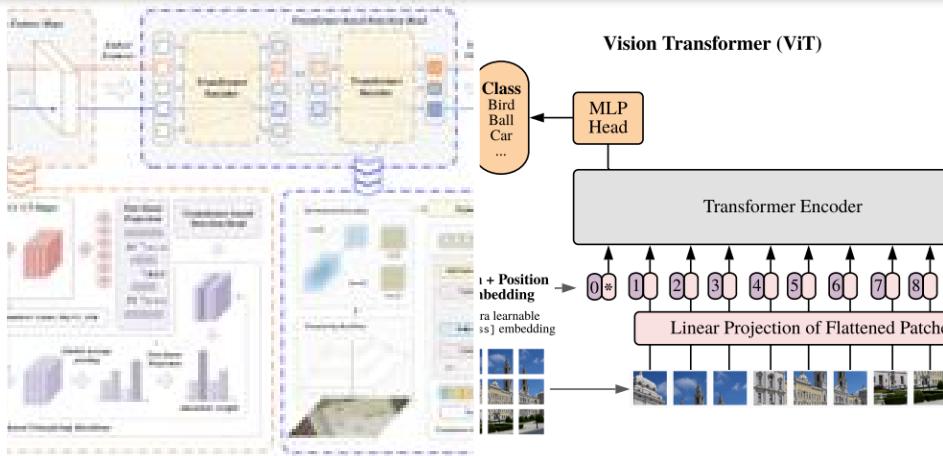
- For pixel-level classification, you can use the model's output to identify regions of interest, then query Stellarium for precise information about specific pixels within these regions.
- This approach combines the efficiency of the ViT model with the accuracy of Stellarium's data.

Remember to carefully curate your dataset, ensuring it covers various celestial conditions and object types. The success of your model will largely depend on the quality and diversity of your training data ⁵.

Share Rewrite







View More

Is a DETR token generated manually (well, by a programmed function) during a training process for these visual transformers?

❖ Sources

[DETR - Hugging Face](#)

[How to Train RT-DETR on a Custom Dataset with Transformers](#)

[DETR: End-to-End Object De](#)

huggingface

blog.roboflow

towardsdatascience

❖ Answer

DETR (DEtection TRansformer) tokens are not manually generated during the training process. Instead, DETR uses learned object queries, which are a fixed set of embeddings that the model learns to associate with different objects in the image ¹. These object queries are part of the model's architecture and are learned end-to-end during training, rather than being manually generated or programmed.

The DETR model consists of a convolutional backbone followed by an encoder-decoder Transformer architecture ¹. The object queries interact with the encoded image features through the decoder to produce the final set of predictions. This approach allows DETR to reason about the relations between objects and the global image context directly ¹.

+ New

⌚ 4m

...



Share

explore different initialization methods for class tokens, including random initialization and pretrained class tokens ⁵. However, these are still learned parameters of the model, not manually generated tokens.

Share Rewrite



Do any of those research papers have linked projects with code on sites like github?

❖ Sources

[PDF] SOIT: Segmenting Objects with Instance-Aware Transformers - AAAI

 cdn.aaai

Git and GitHub Tutorials #6 - Installing Git and Cloning our Repository

 youtube

Instance-Aware Group Quant

 arxiv

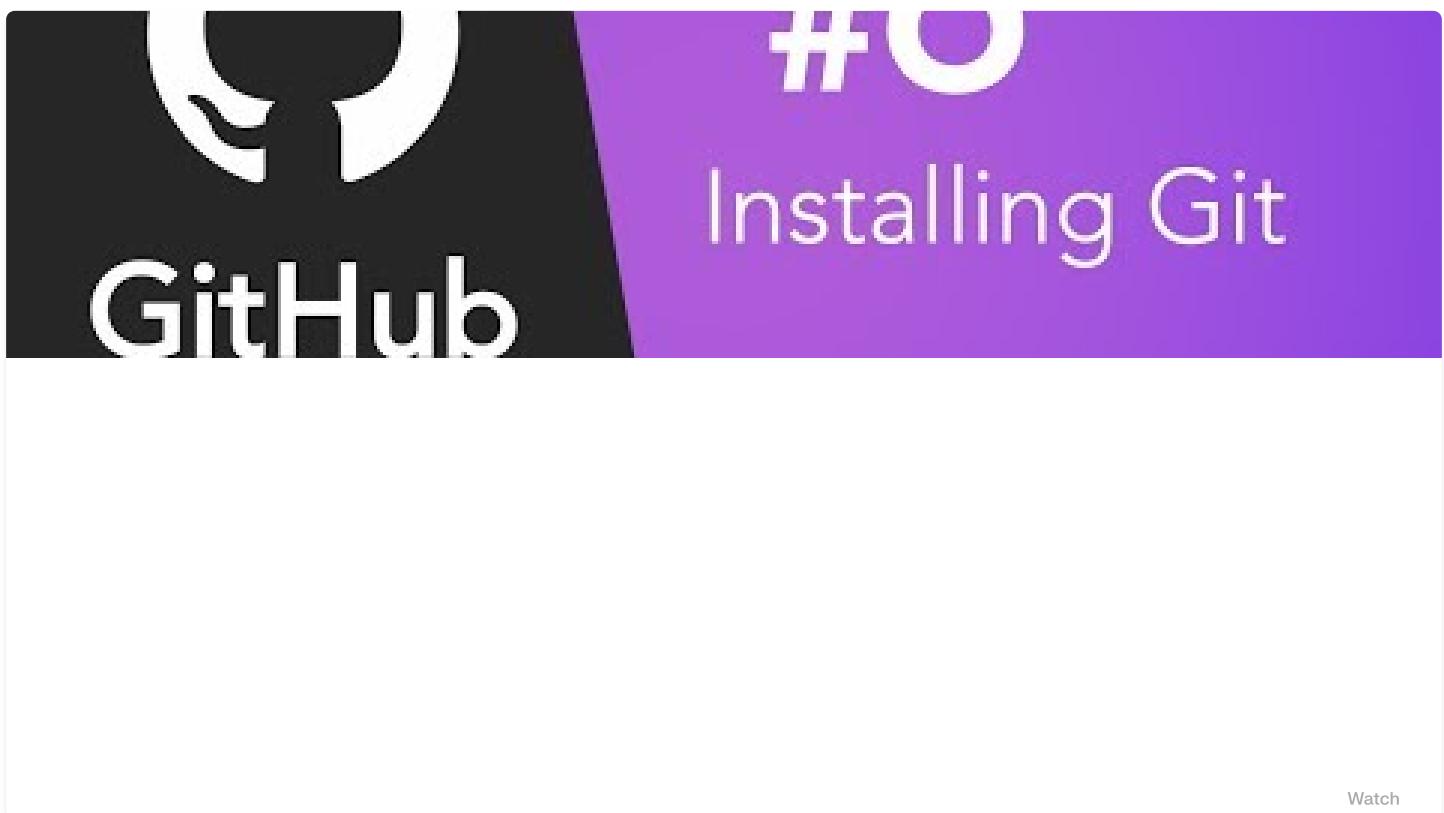
✳️ Answer

Yes, two of the research papers mentioned have linked projects with code:

1. The SOIT (Segmenting Objects with Instance-Aware Transformers) paper has code available on GitHub. The authors state: "Code is available at <https://github.com/yuxiaodongHRI/SOIT>"[1].
2. For the Instance-Aware Group Quantization for Vision Transformers (IGQ-ViT) paper, while there's no direct mention of a GitHub repository, it's noted that the paper is associated with code on Papers with Code ⁷. However, the specific link to the code is not provided in the search results.

 Share  Rewrite





How about the DETR type projects? Do any of those have linked code projects? Any written in Python?

❖ Sources

[facebookresearch/detr: End-to-End Object Detection with Transformers](#)

[github](#)

[DETR: End-to-End Object Detection With Transformers](#)

[alcinos.github](#)

[Object Detection using DETR](#)

[github](#)

❖ Answer

Yes, several DETR (DEtection TRansformer) projects have linked code repositories, and they are primarily written in Python:

1. The original DETR project by Facebook Research is available on GitHub [1](#). It's implemented in PyTorch and includes training code, pretrained models, and a Colab notebook for inference.
2. A custom object detection implementation using DETR is available on GitHub [5](#). This project provides code for fine-tuning DETR on custom datasets and includes installation instructions for required Python libraries.
3. The Deformable DETR project, which improves upon the original DETR, has a GitHub repository [6](#). It's also implemented in Python and aims to address slow convergence issues of the original DETR.
4. Another GitHub repository offers an implementation of DETR for custom object detection tasks [4](#). This project is written in Python and uses PyTorch.

These projects demonstrate that DETR and its variants are actively developed and maintained, with accessible Python implementations available for researchers and practitioners in the field of computer vision and object detection.

Share Rewrite

How much further coding would be required to have DETR output go to both pixel and instance segmentation? Would any of those DETR code samples be more applicable for that goal?

❖ Sources

[Mask Frozen-DETR: High Quality Instance Segmentation with One ...](#)

[openreview](#)

[Panoptic Segmentation: Unifying Semantic and Instance ...](#)

[digitalocean](#)

[Panoptic Segmentation: Unif](#)

[blog.paperspace](#)

❖ Answer

To extend DETR for both pixel-level and instance segmentation, you would need to add a segmentation head to the existing object detection architecture. This process requires some additional coding, but it's not a complete overhaul of the system.

The DetrForSegmentation model, mentioned in the Hugging Face documentation, already combines object detection with panoptic segmentation ⁵ ⁶. This model adds a segmentation mask head on top of DetrForObjectDetection, which can be trained either jointly or in a two-step process:

1. Train DetrForObjectDetection to detect bounding boxes around both "things" (instances) and "stuff" (background elements like trees, roads, sky).
2. Freeze all weights and train only the mask head for about 25 epochs.

To implement this, you would need to:

1. Add a segmentation head to your DETR model.
2. Modify the loss function to include segmentation loss.
3. Adjust the data processing pipeline to include segmentation masks.
4. Update the training loop to handle the new segmentation tasks.

Among the provided code samples, the Hugging Face implementation (DetrForSegmentation) would be the most applicable for this goal ⁵ ⁶. It already includes the necessary architecture for both object detection and panoptic segmentation, which combines instance and semantic segmentation.

If you prefer a more customizable approach, the Mask Frozen-DETR project mentioned in the search results could be relevant ¹. It introduces a framework to convert existing DETR-based object detection models into instance segmentation models by adding a lightweight mask network.

In either case, you'll need to adapt the code to your specific requirements and dataset, but these implementations provide a solid starting point for achieving both pixel-level and instance segmentation with DETR.

[Share](#) [Rewrite](#)

Related

[What are the main differences between Mask Frozen-DETR and Mask DINO](#) +

[How does the training time of Mask Frozen-DETR compare to other instance segmentation models](#) +

[Can Mask Frozen-DETR be integrated with other object detection models](#) +

[What are the key components needed to convert a DETR model into an instance segmentation model](#) +

[Are there any pre-trained models available for Mask Frozen-DETR](#) +



