Could you summarize the main points in the Attention Is All You Need paper?

Sources

Attention Is All You Need - Wikipedia

en.wikipedia

AI: A Comprehensive Guide to 'Attention Is All You Need' in ... - DeepAI

deepai

Decoding "Attention is All You

linkedin

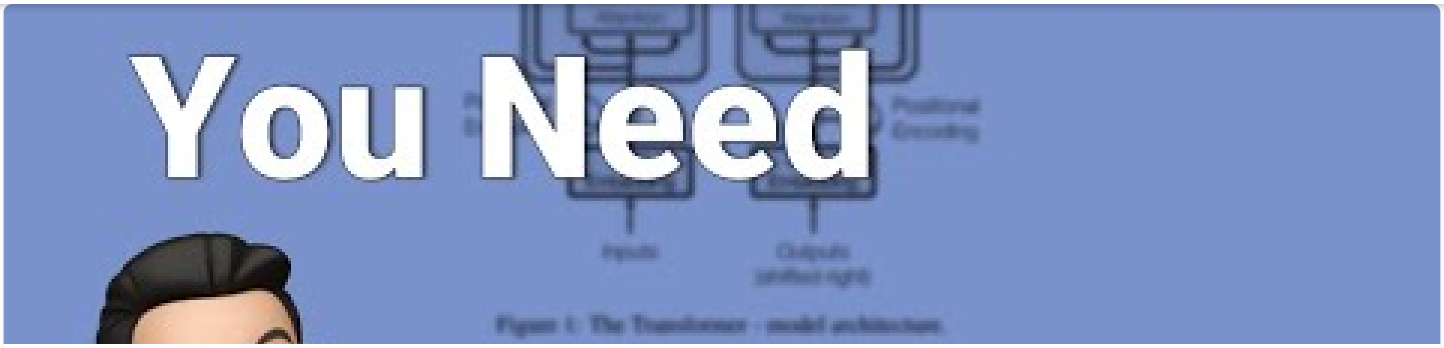
Answer

The "Attention Is All You Need" paper, published in 2017, introduced the Transformer architecture, a groundbreaking approach in machine learning [1] [2]. The main points of the paper are:

1. The Transformer model relies solely on attention mechanisms, eliminating the need for recurrent or convolutional neural networks [4] [8].
2. It achieves superior performance in machine translation tasks while being more parallelizable and requiring less training time [1] [8].
3. The architecture consists of an encoder-decoder structure, both using stacked self-attention and point-wise, fully connected layers [8].
4. Key components of the Transformer include:
   - Self-attention mechanism: Allows each word to attend to all other words in the input sequence [4] [9].
   - Multi-head attention: Enables the model to focus on different aspects of the input simultaneously [1] [8].
   - Positional encoding: Provides information about the position of words in the sequence [1].
5. The model introduces scaled dot-product attention and multi-head attention functions [8].
6. It demonstrates the potential for attention mechanisms to capture long-range dependencies in sequences without regard to their distance [9].

The Transformer architecture has become the foundation for many state-of-the-art models in natural language processing and beyond, revolutionizing the field of artificial intelligence [2] [7].

Share Rewrite



Watch

Are the scaled-dot product attention and multi-head attention functions separate and/or do they occur in a chain with the same input going through one then the next?

Sources

Understanding and Coding Self-Attention, Multi-Head ... - Ahead of AI

magazine.sebastianraschka

Scaled Dot-Product Attention Explained | Papers With Code

paperswithcode

Transformer Multi-Head Atten

blog.gopenai

Answer

Scaled dot-product attention and multi-head attention are related but distinct components of the Transformer architecture. They work together in a specific way:

1. Scaled dot-product attention is a fundamental building block of multi-head attention [2] [3].
2. Multi-head attention uses multiple instances of scaled dot-product attention in parallel [1] [7].

The process works as follows:

1. In multi-head attention, the input is first linearly projected into multiple sets of queries, keys, and values [7].
2. Each set of queries, keys, and values is then processed through its own scaled dot-product attention function [1] [3].
3. The outputs from all the scaled dot-product attention operations are concatenated and linearly projected to produce the final output [7].

This structure allows the model to jointly attend to information from different representation subspaces at different positions [1]. It's analogous to having multiple attention mechanisms running in parallel, each focusing on different aspects of the input sequence [3].

In summary, scaled dot-product attention is a component that operates within each "head" of multi-head attention, rather than being a separate step in a chain [1] [3] [7].

Share Rewrite

In terms of vision transformers, what are MLPs and how do they relate to this setup of multi-head attention, scaled-dot products, and key/query/values?

Sources

Day 12: Multi-Level Perceptron (MLP) and Its Role in LLMs - GoPenAI
blog.gopenai

Multi-Head Attention and Transformers Architecture
kdag-iit-kharagpur.gitbook

Vision Transformer: What It Is
v7labs

Answer

In Vision Transformers (ViTs), Multi-Layer Perceptrons (MLPs) play a crucial role alongside multi-head attention mechanisms. Here's how MLPs relate to the overall architecture:

1. Position in the architecture: MLPs are typically placed after the multi-head attention layer in each transformer block [1] [3]. This sequence allows the model to first process relationships between tokens through attention, then further refine and transform the representations.
2. Function of MLPs:
    - They transform and refine the output from the self-attention layer [1].
    - MLPs add more capacity to the model, enabling it to learn complex patterns [1].
    - They apply transformations independently to each token, allowing for efficient token-wise processing [1].
3. Structure of MLPs in ViTs:
    - MLPs in ViTs are usually two-layer networks with a GELU (Gaussian Error Linear Unit) activation function [3].
    - They often expand the dimensionality of token embeddings before reducing them again, which helps capture more complex patterns [1].
4. MLP head:
    - The final MLP block, also called the MLP head, serves as the output of the transformer [3] [7].
    - It takes the processed features from the transformer encoder and applies additional transformations for task-specific predictions, such as image classification [7].
5. Relationship to attention mechanisms:
    - While multi-head attention focuses on capturing relationships between different parts of the input, MLPs process each token independently [1].
    - This combination allows the model to benefit from both global context (via attention) and local feature refinement (via MLPs).
6. Recent developments:
    - Some researchers are exploring Vision MLPs (V-MLPs) that rely solely on MLPs without self-attention, aiming to achieve global processing through fully connected layers [8].
    - There are also efforts to improve the MLP module in Vision Transformers to reduce computational costs while maintaining accuracy [5].

In summary, MLPs in Vision Transformers complement the multi-head attention mechanism by providing additional non-linear transformations and token-wise processing, contributing to the model's overall ability to learn complex visual representations.

Share Rewrite

In transformers, how much of the key/query/value setup is it necessary for the programmer to specify and how much is learned automatically by the transformer?

Sources

What are Query, Key, and Value in the Transformer Architecture and ...

epichka

Transformers Key-Value Caching Explained - Neptune.ai

neptune

11.1. Queries, Keys, and Values

d2l

Answer

In transformer architectures, the key/query/value setup is largely learned automatically, with minimal direct specification required from the programmer. Here's a breakdown of what is typically specified and what is learned:

Specified by the programmer:

1. Dimensionality: The programmer defines the dimensions for the query, key, and value vectors [1] [2].
2. Weight matrices: The programmer initializes the weight matrices (Wq, Wk, Wv) for the linear projections that create the query, key, and value vectors [1] [2].
3. Architecture: The overall structure of the attention mechanism, including the use of multi-head attention, is defined by the programmer [2].

Learned automatically:

1. Weight values: The actual values of the weight matrices (Wq, Wk, Wv) are learned during training [1] [2].
2. Query, key, and value vectors: These are computed dynamically for each input token using the learned weight matrices [1] [2].
3. Attention patterns: The model learns how to attend to different parts of the input sequence through the training process [1].

The process typically works as follows:

1. The programmer defines the linear layers for query, key, and value projections:

```python
self.W_query = torch.nn.Linear(d_in, d_out, bias=qkv_bias)
self.W_key = torch.nn.Linear(d_in, d_out, bias=qkv_bias)
self.W_value = torch.nn.Linear(d_in, d_out, bias=qkv_bias)
```

2. During the forward pass, the input is transformed into query, key, and value vectors:

```python
queries = self.W_query(x)
keys = self.W_key(x)
values = self.W_value(x)
```

3. The attention mechanism then uses these vectors to compute attention scores and context vectors [2].

In summary, the programmer sets up the framework for the key/query/value mechanism, but the transformer learns how to effectively use this mechanism through training on large datasets. This allows the model to adaptively focus on relevant parts of the input for different tasks and contexts.

Share Rewrite

So the key/query/value are all just differently-sized areas in the hidden layers that are learned during training so that they get associated with different inputs? And they're setup that way in order that there's something to use with vector databases?

Sources

Multi-head attention mechanism: "queries", "keys", and "values ... — data-science-blog

Attention in Transformers Query, Key and Value in Machine Learning — youtube

Transformers Explained Visua — towardsdatascience

Answer

Your understanding is partially correct, but there are some important clarifications to make:

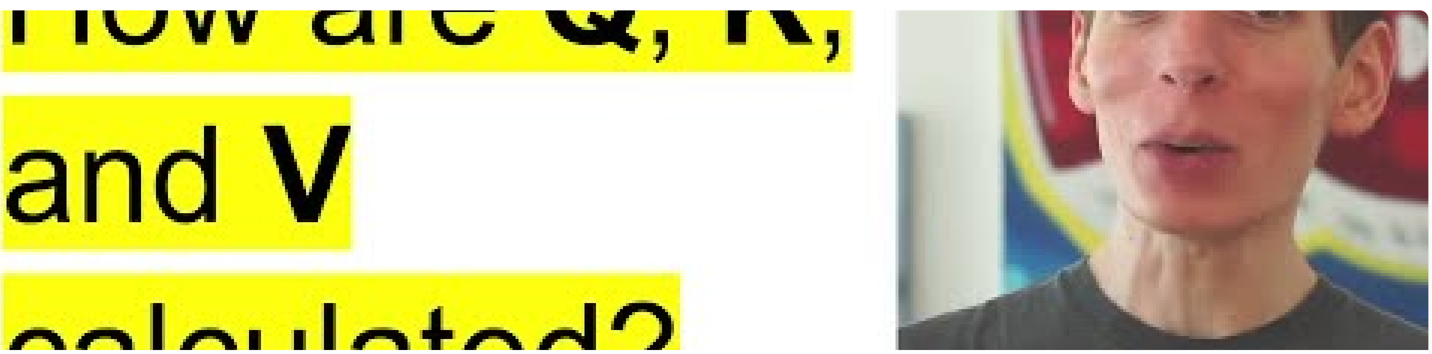
1. Key, query, and value vectors are not pre-defined areas in hidden layers. They are derived from the input through learned linear transformations [1] [3].
2. These vectors are not directly associated with different inputs. Instead, they are computed for each input token in the sequence [2].
3. The key, query, and value concept wasn't specifically designed for vector databases. It's a fundamental part of the attention mechanism in transformer models [5].

The process works as follows:

1. The input sequence is first embedded into a continuous vector space [3].
2. For each token, three separate linear transformations are applied to create the query, key, and value vectors [3] [4].
3. These transformations are learned during training, allowing the model to adapt how it computes attention for different tasks [5].
4. The attention mechanism then uses these vectors to determine how much each token should attend to other tokens in the sequence [1] [6].

The key/query/value concept in transformers is primarily used for self-attention within the model, enabling it to focus on relevant parts of the input when processing each token. While this concept can be useful for vector databases, it wasn't originally designed for that purpose.

Share Rewrite



Watch

How does it learn what a token is? For instance in language, tokens can be sub-parts of words, so much that I can't tell what the tokens will be when looking at any arbitrary word in a sentence. For visual transformers, what would tokens be? They are learned, right? Or is it something specified by a feature detector?

Sources

Improving Vision Transformer Efficiency and Accuracy by Learning ...

research

Efficient Visual Transformer by Learnable Token Merging - arXiv

arxiv

Tokens-to-Token Vision Trans

towardsdatascience

Answer

In visual transformers, tokens are learned representations of image patches, but the process of tokenization is different from language models. Here's how it works:

1. Initial tokenization: The image is typically divided into fixed-size patches (e.g., 16x16 pixels). These patches serve as the initial tokens [1] [4].
2. Learnable embeddings: Each patch is then transformed into a learnable embedding vector using a linear projection. This projection is learned during training [1].
3. Token refinement: Some advanced approaches, like Tokens-to-Token ViT (T2T-ViT), use a progressive tokenization process. This involves:
   - Restructuring tokens into image-like grids
   - Splitting these grids with overlapping
   - Aggregating neighboring tokens [5]
4. Adaptive tokenization: Recent research, such as TokenLearner, proposes methods to adaptively generate a smaller number of tokens based on the input image, rather than using fixed uniform splitting [1].
5. Learnable Token Merging (LTM): This approach learns to merge tokens in a data-driven manner, reducing the number of tokens while preserving important information [2].

Unlike in language models where tokens often correspond to linguistic units, visual tokens represent learned features of image patches. These features are not predefined or based on specific feature detectors but are learned end-to-end during the training process. The transformer architecture then learns to attend to these tokens to perform various vision tasks.

The key difference from language models is that in vision transformers, the initial tokenization is typically based on fixed-size patches, but the representation of these patches (i.e., the actual "tokens" processed by the transformer) is learned. The model learns what visual information is important to represent within each token through the training process.

Share Rewrite

As a programmer, I still need to perform some amount of supervised training on a visual transformer, right? Today for instance they're probably still using YOLO-like bounding boxes or similar.

Sources

Vision Transformer Model: Architecture, development and applications
leewayhertz

Object detection - Hugging Face
huggingface

Building a Vision Transformer
akshaymakes

Answer

Yes, as a programmer, you still need to perform supervised training on a visual transformer for tasks like object detection. While Vision Transformers (ViTs) have introduced new architectures, they still rely on supervised learning techniques similar to traditional convolutional neural networks.

For object detection tasks, many current approaches still utilize YOLO-like bounding box representations:

1. YOLO-style bounding boxes are typically represented as [center_x, center_y, width, height] normalized to the range [0, 1] [4] [8].
2. Some models, like DETR (Detection Transformer), expect bounding boxes in this YOLO format during preprocessing and evaluation [10].
3. The training process for ViTs in object detection tasks often involves:
   - Preparing a labeled dataset with images and corresponding bounding box annotations [1].
   - Preprocessing the input images and annotations to match the model's expected format [1] [2].
   - Training the model using techniques like minimizing a loss function (e.g., cross-entropy) to predict accurate bounding boxes and class labels [1] [5].
4. Recent developments have explored ways to adapt transformers for object detection with minimal modifications. For example, the YOLOS (You Only Look at One Sequence) model attempts to perform object detection using a vanilla Vision Transformer architecture with few changes [9].

While ViTs have introduced new ways of processing image data, the core concept of supervised learning with labeled bounding boxes remains crucial for training object detection models, including those based on transformer architectures.

Share Rewrite

Is there any way to have a trained transformer increase the accuracy of the training bounding boxes such that it gets closer to pixel level?

Sources

Bounding Box Explained in AI | Restackio
restack

[PDF] End-to-End Object Detection with Transformers
ecva

A Review of Transformer-Base
arxiv

Answer

Yes, there are several approaches to improve the accuracy of bounding boxes in transformer-based models, moving towards pixel-level precision:

1. Mask DINO: This unified transformer-based framework aims to improve pixel-level alignment for detection and segmentation tasks. It builds upon the DINO (DETR with Improved DeNoising Anchor Boxes) architecture to achieve better feature alignment between detection and segmentation [6].
2. SOF-DETR (Small Object-Focused DETR): This model focuses on improving small object detection using transformers. It employs a specialized loss function to enhance the accuracy of bounding boxes, particularly for smaller objects [7].
3. High-Resolution Vision Transformers: Inspired by super-resolution architectures, these models learn to resize high-resolution images and masks, retaining valuable local information. This approach can lead to more precise pixel-level identification [8].
4. Local Information Capture Module (LICM): Proposed as part of the SENet architecture, this module enhances the transformer's ability to model local information, which is crucial for pixel-level binary segmentation tasks [4].
5. Multi-Scale Prediction: This technique involves using different scales of input images to improve the detection of objects of varying sizes, particularly small objects. It can help in refining bounding box accuracy [1].
6. Deformable DETR: This variant incorporates a deformable mechanism into self-attention, addressing convergence issues and potentially improving bounding box precision [1].

These approaches demonstrate that transformer-based models can indeed be adapted to increase bounding box accuracy, moving closer to pixel-level precision. However, it's important to note that achieving perfect pixel-level accuracy remains a challenging task, and ongoing research continues to push the boundaries in this area.

Share Rewrite

How is a transformer with ViT additions different from DETR?

Sources

What is Detection Transformers (DETR)? - Zilliz Learn

zilliz

What is DETR (Detection Transformers)? - Roboflow Blog

blog.roboflow

DETR: Overview and Inference

learnopencv

Answer

Vision Transformers (ViT) and Detection Transformers (DETR) are both transformer-based architectures for computer vision tasks, but they differ in several key aspects:

1. Purpose and output:
   - ViT: Primarily designed for image classification tasks.
   - DETR: Specifically developed for object detection, predicting bounding boxes and class labels for objects in an image [1].
2. Input processing:
   - ViT: Divides the input image into fixed-size patches, which are then linearly embedded.
   - DETR: Uses a CNN backbone (typically ResNet) to extract initial features from the image [3].
3. Decoder structure:
   - ViT: Does not use a transformer decoder.
   - DETR: Incorporates a transformer decoder, which is a key innovation. It takes encoded features and learned object queries as inputs [3].
4. Object queries:
   - ViT: Does not use object queries.
   - DETR: Introduces learnable object queries, which are used to predict object locations and classes [1] [3].

5. Output prediction:
   - ViT: Typically uses a single classification head.
   - DETR: Employs prediction heads (feed-forward networks) to output bounding boxes and classes for detected objects [1] [3].
6. Positional encoding:
   - ViT: Uses positional embeddings for image patches.
   - DETR: Adds positional encodings to the CNN encoder output to provide spatial information [2] [3].
7. End-to-end approach:
   - ViT: Focuses on general feature learning for classification.
   - DETR: Provides an end-to-end object detection pipeline, eliminating the need for hand-designed priors or post-processing steps like non-maximum suppression [3].

In summary, while both ViT and DETR leverage transformer architectures for computer vision tasks, DETR is specifically tailored for object detection with additional components like object queries and a transformer decoder, whereas ViT is more focused on general image feature learning for classification tasks.

Share Rewrite

OK, I'm going to use one of those or a combination of the two in order to train my desired transformer setup. I want to train a longer range object detector for night sky usage that filters out stars and planets. I can use difference of gaussians and Stellarium to generate the training data / feature detection. For instance, I'm going to give Stellarium commands such as location, camera vector, and time to generate thousands of images. Then, I need some way to query Stellarium for object names within images. It won't be exactly the camera vector, but a slight offset to aim at light sources within those generated images. I need a way to figure out what the angle offsets from the camera center vector are for any pixel position. I'll find the pixel positions automatically with difference of gaussians.

Sources

An improved method for object detection in astronomical images — academic.oup

Difference of Gaussians Edge Enhancement Algorithm — olympus-lifescience

Automatic Space Object Dete — dorazaria.github

Answer

To create a long-range object detector for night sky usage that filters out stars and planets, you can combine several techniques:

1. Data Generation:
   - Use Stellarium to generate thousands of night sky images with varying locations, camera vectors, and times.
   - Implement the Difference of Gaussians (DoG) algorithm to detect light sources in these images [4] [6].
2. Object Identification:
   - Query Stellarium for object names within the generated images.
   - Use a slight offset from the camera vector to aim at detected light sources.
3. Angle Offset Calculation:
   - To calculate angle offsets from the camera center vector for any pixel position:
     a. Determine the camera's field of view (FoV).
     b. Calculate the angular resolution (degrees per pixel).
     c. Compute the offset using the pixel's distance from the center.
4. Training Data Preparation:
   - Label non-stellar objects detected by DoG.
   - Create a dataset with images, object locations, and labels.

5. Transformer Setup:
    - Use a Vision Transformer (ViT) architecture, modifying it for object detection [1] [3].
    - Implement a DETR-like approach, using object queries to predict bounding boxes and classes [3].
6. Training:
    - Train the model to detect non-stellar objects while ignoring stars and planets.
    - Use the generated dataset with labeled objects and their angle offsets.
7. Performance Optimization:
    - Consider using SWIR (Short-Wave Infrared) imaging for improved long-range detection [7].
    - Implement adaptive noise reduction techniques for better detection of faint objects [1].

By combining these techniques, you can create a transformer-based model capable of detecting and tracking non-stellar objects in the night sky while filtering out stars and planets.

Share Rewrite

Could you write a function that takes in camera FoV and resolution and outputs the angle differences (one up/down & one left/right) per pixel? For instance, if I give it the camera FoV parameters and an (x,y) pixel position, it should return angle horizontal and angle vertical from the center of the image (where I'm assuming the camera vector will be pointing at).

Sources

finding relative angles - OpenMV Forums
forums.openmv

Calculate angle from camera to detected object : r/computervision
reddit

Camera Field of View Calculat
omnicalculator

Answer

Here's a Python function that calculates the angle differences (horizontal and vertical) for a given pixel position based on the camera's field of view (FoV) and resolution. This function assumes that the camera's center vector points at the center of the image:

python

```python
import math

def calculate_angle_offsets(fov_horizontal, fov_vertical, resolution_width, resolution_height, pixel_x, pixel_y):
    # Calculate the angle per pixel
    angle_per_pixel_horizontal = fov_horizontal / resolution_width
    angle_per_pixel_vertical = fov_vertical / resolution_height

    # Calculate the center of the image
    center_x = resolution_width / 2
    center_y = resolution_height / 2

    # Calculate the pixel offsets from the center
    offset_x = pixel_x - center_x
    offset_y = pixel_y - center_y

    # Calculate the angles from the center
    angle_horizontal = offset_x * angle_per_pixel_horizontal
    angle_vertical = offset_y * angle_per_pixel_vertical

    return angle_horizontal, angle_vertical

# Example usage:
```

```
fov_horizontal = 70.8  # Horizontal FoV in degrees
fov_vertical = 55.6     # Vertical FoV in degrees
resolution_width = 320   # Image width in pixels
resolution_height = 240  # Image height in pixels

pixel_x = 20  # Example pixel x-coordinate
pixel_y = 25  # Example pixel y-coordinate

angles = calculate_angle_offsets(fov_horizontal, fov_vertical, resolution_width, resolution_height, pixel_x, 
pixel_y)
print(f"Angle Horizontal: {angles[0]:.4f} degrees, Angle Vertical: {angles[1]:.4f} degrees")
```

Explanation:

- **Input Parameters**:
  - `fov_horizontal`: The horizontal field of view in degrees.
  - `fov_vertical`: The vertical field of view in degrees.
  - `resolution_width`: The width of the image in pixels.
  - `resolution_height`: The height of the image in pixels.
  - `pixel_x`: The x-coordinate of the pixel for which you want to calculate the angle.
  - `pixel_y`: The y-coordinate of the pixel for which you want to calculate the angle.
- **Calculations**:
  - The function first calculates how many degrees each pixel represents by dividing the total FoV by the respective resolution dimensions.
  - It then finds the offsets of the specified pixel from the center of the image.
  - Finally, it computes the angles by multiplying these offsets by their respective angle-per-pixel values.

This function will return the horizontal and vertical angles from the camera's perspective for any given pixel position within an image.

Share Rewrite

Can you write a function to make a new vector taking the center vector and adding those two angles?

Answer

Certainly! Here's a Python function that takes a center vector and adds the horizontal and vertical angles to create a new vector:

python

```python
import math

def create_new_vector(center_vector, angle_horizontal, angle_vertical):
    # Unpack the center vector
    cx, cy, cz = center_vector

    # Convert angles from degrees to radians
    angle_h_rad = math.radians(angle_horizontal)
    angle_v_rad = math.radians(angle_vertical)

    # Calculate the magnitude of the center vector
    magnitude = math.sqrt(cx**2 + cy**2 + cz**2)
```

```python
    # Normalize the center vector
    nx, ny, nz = cx/magnitude, cy/magnitude, cz/magnitude

    # Rotate around the vertical axis (for horizontal angle)
    rx = nx * math.cos(angle_h_rad) - nz * math.sin(angle_h_rad)
    ry = ny
    rz = nx * math.sin(angle_h_rad) + nz * math.cos(angle_h_rad)

    # Rotate around the horizontal axis (for vertical angle)
    fx = rx
    fy = ry * math.cos(angle_v_rad) - rz * math.sin(angle_v_rad)
    fz = ry * math.sin(angle_v_rad) + rz * math.cos(angle_v_rad)

    # Normalize the final vector
    magnitude = math.sqrt(fx**2 + fy**2 + fz**2)
    new_vector = (fx/magnitude, fy/magnitude, fz/magnitude)

    return new_vector

# Example usage:
center_vector = (0, 0, 1)  # Assuming the camera is pointing along the z-axis
angle_horizontal = 5  # 5 degrees to the right
angle_vertical = -3  # 3 degrees down

new_vector = create_new_vector(center_vector, angle_horizontal, angle_vertical)
print(f"New vector: {new_vector}")
```

This function does the following:

1. It takes three parameters:
    - `center_vector`: A tuple (x, y, z) representing the direction the camera is pointing.
    - `angle_horizontal`: The horizontal angle offset in degrees.
    - `angle_vertical`: The vertical angle offset in degrees.
2. It normalizes the center vector to ensure we're working with a unit vector.
3. It applies two rotations:
    - First, it rotates around the vertical axis (y-axis in this case) to account for the horizontal angle.
    - Then, it rotates around the horizontal axis (x-axis after the first rotation) to account for the vertical angle.
4. Finally, it normalizes the resulting vector to ensure it's a unit vector.

This function assumes a right-handed coordinate system where:

- The x-axis points to the right
- The y-axis points up
- The z-axis points forward (into the scene)

You can adjust the axes if your system uses a different convention. Also, note that this function assumes small angles. For very large angles, you might need to use a more complex rotation method, such as quaternions.

Can you write a function to take the output vector from the above function and query Stellarium for what object is at that vector? The function should take a datetime, lat/long observation location, and that vector. It should output an object type (if any) and the name of the object. If Stellarium returns that nothing is there, the return type should be none.

Sources

k96e/StellariumRC: Control Stellarium by Python - GitHub

github

yoavnavon/stellarium-python-client - GitHub

github

Making videos of the night sk

beltoforion

Answer

Here's a Python function that queries Stellarium for an object at a given vector, using the StellariumRC library:

python

```python
from datetime import datetime
import StellariumRC

def query_stellarium_object(observation_datetime, latitude, longitude, vector):
    s = StellariumRC.Stellarium()

    # Set the observation time and location
    s.core.setDate(observation_datetime.strftime("%Y-%m-%d %H:%M:%S"))
    s.core.setObserverLocation(longitude, latitude, 0, 1, "", "Earth")

    # Convert vector to altitude and azimuth
    alt_az = s.core.vector2altAzi(vector[0], vector[1], vector[2])

    # Query Stellarium for the object at the given position
    object_info = s.core.getObjectInfo(alt_az[0], alt_az[1], "")

    if object_info:
        return object_info['type'], object_info['name']
    else:
        return None, None

# Example usage:
observation_time = datetime(2025, 1, 11, 19, 0, 0)  # January 11, 2025, 7 PM
latitude = 41.8781  # Chicago latitude
longitude = -87.6298  # Chicago longitude
vector = (0.1, 0.2, 0.9)  # Example vector

object_type, object_name = query_stellarium_object(observation_time, latitude, longitude, vector)

if object_type and object_name:
    print(f"Object found: {object_type} - {object_name}")
else:
    print("No object found at the specified position.")
```

This function does the following:

1. It initializes a connection to Stellarium using the StellariumRC library.
2. It sets the observation time and location in Stellarium using the provided datetime and latitude/longitude.
3. It converts the input vector to altitude and azimuth coordinates using Stellarium's `vector2altAzi` method.
4. It queries Stellarium for the object at the given position using the `getObjectInfo` method.
5. If an object is found, it returns the object type and name. If no object is found, it returns `None` for both type and name.

Note that this function assumes Stellarium is running and the RemoteControl plugin is enabled. Also, make sure to install the StellariumRC library using `pip install stellariumrc` before running this code 1 .

Share Rewrite

Can you write a function that scans over an image and returns a list of (x,y) coordinates for the pixels that get positive results for a difference of gaussians? I'd be looking for point source lights like stars or possibly planets. I think it might take an argument for brightness difference, not sure.

Answer

Certainly! Here's a Python function that uses the Difference of Gaussians (DoG) method to detect point sources of light in an image, such as stars or planets. This function uses OpenCV and NumPy libraries for image processing:

python

```python
import cv2
import numpy as np

def detect_point_sources(image_path, low_sigma=1, high_sigma=2, threshold=10):
    # Read the image in grayscale
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Apply Gaussian blurs
    low_blur = cv2.GaussianBlur(img, (0, 0), low_sigma)
    high_blur = cv2.GaussianBlur(img, (0, 0), high_sigma)

    # Compute the Difference of Gaussians
    dog = low_blur - high_blur

    # Threshold the DoG image
    _, binary = cv2.threshold(dog, threshold, 255, cv2.THRESH_BINARY)

    # Find contours
    contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Get centroids of contours
    point_sources = []
    for contour in contours:
        M = cv2.moments(contour)
        if M["m00"] != 0:
            cX = int(M["m10"] / M["m00"])
            cY = int(M["m01"] / M["m00"])
            point_sources.append((cX, cY))

    return point_sources

# Example usage:
image_path = "night_sky.jpg"
point_sources = detect_point_sources(image_path, low_sigma=1, high_sigma=2, threshold=10)
print(f"Detected {len(point_sources)} point sources:")
for x, y in point_sources:
    print(f"  ({x}, {y})")
```

This function does the following:

1. It takes four parameters:
    - `image_path`: The path to the input image file.
    - `low_sigma`: The standard deviation for the first Gaussian blur (default: 1).
    - `high_sigma`: The standard deviation for the second Gaussian blur (default: 2).

- `threshold`: The brightness threshold for detecting point sources (default: 10).

2. It reads the image in grayscale.
3. It applies two Gaussian blurs with different sigma values.
4. It computes the Difference of Gaussians by subtracting the more blurred image from the less blurred one.
5. It applies a threshold to the DoG image to create a binary image.
6. It finds contours in the binary image.
7. For each contour, it calculates the centroid and adds it to the list of point sources.
8. Finally, it returns a list of (x, y) coordinates for the detected point sources.

You can adjust the `low_sigma`, `high_sigma`, and `threshold` parameters to fine-tune the detection sensitivity. Lower threshold values will detect more point sources but may also increase false positives.

This function provides a good starting point for detecting stars and planets in night sky images. However, for more advanced applications, you might want to consider additional filtering techniques or machine learning approaches to improve accuracy and reduce false positives.

Share Rewrite

Let's brainstorm the training process for the potential transformer/DETR/ViT (whatever we end up with). We'll have performed a difference of gaussians on a single Stellarium output image (that image will have had the datetime, observation lat/long, and camera vector in the name) and gotten a list of points of light. Then we'll have queried Stellarium for possible names of objects at those positions. Now we need to give the transformer the original image and use that list in some way to calculate error. We'll need to expect certain things in the output layer. I guess we could have a class for every one of the potential planets and stars that are human-eye/camera with no magnification visible. That should put it in the range of 5000-6000 classes. I'd like the transformer to learn the classes. I hope it can indicate x,y positions and the class number. I'm not sure what it should do, if anything, to unknown objects.

Answer

Your approach sounds promising. Let's break down the training process and consider how to structure the transformer's input and output:

1. Input:
   - Original Stellarium image
   - Metadata (datetime, observation lat/long, camera vector) encoded as additional input features
2. Output layer:
   - Object detection heads (similar to DETR):
     - Bounding box coordinates (x, y, width, height) or just (x, y) for point sources
     - Class prediction (5000-6000 classes for known celestial objects)
     - Objectness score (confidence that it's a valid detection)
3. Training process:
   a. Prepare dataset:
   - Generate Stellarium images with metadata
   - Perform DoG to get points of light
   - Query Stellarium for object names at those positions
   - Create ground truth labels: (x, y, class_id) for each detected object

   b. Model architecture:
   - Use a ViT-like backbone for feature extraction
   - Add DETR-like object queries and transformer decoder

- Output heads for bounding box, class prediction, and objectness score

c. Loss function:

- Bounding box loss (e.g., L1 loss)
- Classification loss (e.g., cross-entropy)
- Objectness loss (e.g., binary cross-entropy)

d. Training loop:

- Feed images and metadata through the model
- Compare predictions with ground truth
- Backpropagate and update weights

4. Handling unknown objects:
   - Add an "unknown" class
   - Train the model to predict this class for objects not in the known set
   - Use the objectness score to determine if it's a valid detection
5. Post-processing:
   - Non-maximum suppression to remove duplicate detections
   - Filtering based on objectness score
6. Evaluation:
   - Use metrics like mAP (mean Average Precision) for object detection
   - Confusion matrix for classification accuracy

This approach combines the strengths of ViT for feature extraction and DETR for object detection, while being tailored to your specific night sky observation task.

Share Rewrite

What is "objectness loss" and how does it differ from classification loss?

Sources

YOLO V3 Explained - Towards Data Science
towardsdatascience

Yolo Loss function explanation - Cross Validated - Stack Exchange
stats.stackexchange

How does YOLO loss work? -
educative

Answer

Objectness loss and classification loss are two distinct components of object detection loss functions:

Objectness loss:

- Measures the model's ability to predict the presence of an object in a bounding box [1] [4].
- Teaches the network to predict a correct IoU (Intersection over Union) between the predicted box and the ground truth [1].
- Penalizes the model for incorrect confidence scores about object presence [3] [5].

Classification loss:

- Calculates the difference in probabilities between the actual and predicted object classes [3].
- Penalizes the model for misclassifying the type of object detected [5].
- Only applies to boxes that are determined to contain objects [2].

Key differences:

1. Purpose: Objectness loss focuses on detecting object presence, while classification loss deals with identifying the specific type of object.
2. Application: Objectness loss applies to all predicted boxes, whereas classification loss only affects boxes containing objects [2].
3. Prediction target: Objectness loss predicts IoU or confidence scores, while classification loss predicts class probabilities [1] [3].

Both losses work together in object detection models to improve overall performance in locating and identifying objects in images.

Share Rewrite

That's interesting. So the model might start to get better at classifying star types from other star types or stars from planets?

Answer

Yes, exactly! The combination of objectness loss and classification loss in the training process can lead to several interesting improvements in the model's performance:

1. Star vs. Planet Differentiation:
   - The model can learn to distinguish between stars and planets based on subtle visual cues, such as the steadiness of light (planets don't twinkle like stars) or slight color differences.
2. Star Type Classification:
   - As the model processes more images, it can start to differentiate between various star types (e.g., red giants, white dwarfs, main sequence stars) based on their color and brightness.
3. Celestial Object Identification:
   - The model can learn to identify specific well-known celestial objects like Polaris, Sirius, or Jupiter, improving its ability to recognize these consistently across different images.
4. Handling Variability:
   - It might become better at handling variable stars, recognizing their changing brightness patterns over time.
5. Constellation Recognition:
   - While not explicitly trained for this, the model might indirectly learn to recognize patterns of stars that form constellations.
6. Artifact Rejection:
   - The model could improve at distinguishing actual celestial objects from artifacts like lens flares or satellite trails.
7. Faint Object Detection:
   - As it learns to better classify bright objects, it might also improve at detecting and classifying fainter objects that are harder to distinguish.

This learning process allows the model to develop a more nuanced understanding of the night sky, potentially leading to more accurate and detailed classifications over time.

Share Rewrite

But we're dealing with single images at a time here, so steadiness of light might not be a factor, right?

Answer

You're absolutely correct, and I apologize for that oversight. You're right that steadiness of light (twinkling) wouldn't be a factor when dealing with single images. Let me clarify and provide more relevant distinctions:

1. Appearance in Single Images:
   - Stars typically appear as perfect point sources of light.
   - Planets, while still small, might appear as tiny discs with a more defined edge.
2. Color and Brightness:
   - Stars have a wide range of colors depending on their temperature.
   - Planets often have distinct colors (e.g., Mars appears reddish, Jupiter slightly orange).
3. Relative Brightness:
   - Planets like Venus, Jupiter, and Mars can appear brighter than many stars.
   - The relative brightness of planets compared to surrounding stars can be a distinguishing factor.
4. Position:
   - Planets are found along the ecliptic (the plane of the solar system).
   - Stars maintain fixed positions relative to each other.
5. Resolution and Detail:
   - With sufficient resolution, planets might show some surface features or atmospheric patterns.
   - Stars remain point sources regardless of image resolution.

The model would learn these static visual cues to differentiate between stars, planets, and other celestial objects in single images. Thank you for pointing out this important distinction – it's crucial for accurate model training and expectations.

Share Rewrite

If you were looking at the sky with a simple camera with only 1080p resolution, would there be a difference between eye-visible stars and a planet like Venus?

Sources

Planetary Imaging with Your DSLR Camera - Sky & Telescope

skyandtelescope

ESA - Venus setting - European Space Agency

esa

Best telescopes for seeing pla

space

Answer

When observing the night sky with a 1080p camera, there would be noticeable differences between eye-visible stars and a planet like Venus:

1. Appearance:
   - Stars would appear as tiny point sources of light, essentially occupying a single pixel or a small cluster of pixels.
   - Venus, being much closer and larger in apparent size, would likely appear as a small but distinct disc, potentially spanning several pixels [1] [5].
2. Brightness:
   - Venus is often one of the brightest objects in the night sky, typically appearing brighter than most stars [4].
   - This brightness difference would be evident in a 1080p image, with Venus potentially appearing as a more prominent, brighter spot.

3. Color:
    - Stars may show subtle color variations depending on their temperature, though this might be difficult to discern in a basic 1080p image.
    - Venus often appears as a bright, white or slightly yellowish object, which could be noticeable even in a lower-resolution image [4].
4. Detail:
    - Stars would remain unresolved points of light.
    - Venus, while not showing surface details, might display a perceptible disc shape or at least appear larger than surrounding stars [1] [5].
5. Position:
    - Venus would be found along the ecliptic (the plane of the solar system), which could help distinguish it from stars if the observer is familiar with celestial positioning [4].

While a 1080p camera without additional magnification or specialized equipment would not reveal significant details of Venus's surface or phases, it should still be possible to distinguish the planet from stars based on its brightness, apparent size, and possibly its color [1] [4] [5].

Share Rewrite

OK, this means that I need to go back over my difference of gaussian approach. If I'm trying to calculate a bounding box / if Venus (or potentially other planets) take up multiple pixels, I need to readdress that function or add another function before the next one in order to generate a bounding box.

Answer

You're absolutely right. Given that planets like Venus might occupy multiple pixels in a 1080p image, we should modify our approach to accommodate both point sources (stars) and slightly larger objects (planets). Here's an updated strategy:

1. Modify the Difference of Gaussians function:

```python
import cv2
import numpy as np

def detect_celestial_objects(image_path, low_sigma=1, high_sigma=2, threshold=10, min_area=1, max_area=50):
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    low_blur = cv2.GaussianBlur(img, (0, 0), low_sigma)
    high_blur = cv2.GaussianBlur(img, (0, 0), high_sigma)

    dog = low_blur - high_blur

    _, binary = cv2.threshold(dog, threshold, 255, cv2.THRESH_BINARY)

    contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    celestial_objects = []
    for contour in contours:
        area = cv2.contourArea(contour)
        if min_area <= area <= max_area:
            x, y, w, h = cv2.boundingRect(contour)
            celestial_objects.append((x, y, w, h))

    return celestial_objects
```

```python
# Example usage:
image_path = "night_sky.jpg"
celestial_objects = detect_celestial_objects(image_path, min_area=1, max_area=50)
print(f"Detected {len(celestial_objects)} celestial objects:")
for x, y, w, h in celestial_objects:
    print(f"  Bounding box: ({x}, {y}, {w}, {h})")
```

This updated function:

- Uses `cv2.boundingRect()` to get bounding boxes for detected objects.
- Includes `min_area` and `max_area` parameters to filter out very small (noise) or very large (non-celestial) objects.
- Returns a list of tuples (x, y, width, height) for each detected object.

2. Adjust the Stellarium query function:
   You'll need to modify the `query_stellarium_object` function to accept a bounding box instead of a single point. You might want to query Stellarium for the center point of the bounding box, or potentially for multiple points within the box for larger objects.
3. Update the training process:

- Use the bounding box information in your loss function calculations.
- Consider using IoU (Intersection over Union) as part of your objectness loss.

4. Model output:
   Ensure your model is designed to output bounding boxes (x, y, width, height) instead of just (x, y) coordinates.

This approach will allow your model to handle both point-like stars and slightly larger objects like visible planets, providing more accurate detections and classifications.

Share Rewrite

Does Stellarium have any functionality for querying for all objects in a bounding box? Doubt it. Maybe.

Sources

BBOX and GEOMETRY query structure - Forum@Sentinel Hub
forum

[PDF] Stellarium 24.2 User Guide
mirror.umd

[PDF] Stellarium 24.4 User Gu (editors ...
stellarium

Answer

Based on the available search results, there is no direct mention of Stellarium having functionality to query all objects within a bounding box. However, Stellarium does have some related features that might be useful:

1. Stellarium has a Search Window with various tabs for finding objects [2] [3].
2. The software includes a "CAbstractAPIService" class that provides predefined location search functionality [4].
3. Stellarium has a feature to display a red bounding box of the area that will be captured when taking screenshots [7].

While these features suggest Stellarium has some search and bounding box capabilities, they don't specifically indicate the ability to query all objects within a given bounding box. It's possible this functionality exists but isn't mentioned in the provided search results. To get a definitive answer, you may need to consult the full Stellarium documentation or ask in their community forums.

Share Rewrite

OK, we still can use the bounding box addition to the difference of gaussian function, but we'll need to come up with a strategy to query Stellarium using it. If the bounding box is a single pixel in size then we can go directly to the next function. If it's larger, we

might need to query Stellarium multiple times.

Sources

Get bounding box for asset through Cesium ion REST API

community.cesium

RemoteControl plugin HTTP API description - Stellarium

stellarium

Is it possible to query via bou
API?

gis.stackexchange

Answer

To handle both single-pixel stars and larger objects like planets in our Stellarium queries, we can implement the following strategy:

1. For single-pixel objects (bounding box width and height = 1):
   - Use the existing `query_stellarium_object` function with the center coordinates.
2. For larger objects (bounding box width or height > 1):
   - Query Stellarium for the center point of the bounding box.
   - If no object is found, query the corners of the bounding box.
   - If still no object is found, perform a grid search within the bounding box.

Here's a modified version of the `query_stellarium_object` function to implement this strategy:

python

```python
def query_stellarium_objects(s, observation_datetime, latitude, longitude, bbox):
    x, y, width, height = bbox
    center_x, center_y = x + width // 2, y + height // 2

    # Set the observation time and location
    s.core.setDate(observation_datetime.strftime("%Y-%m-%d %H:%M:%S"))
    s.core.setObserverLocation(longitude, latitude, 0, 1, "", "Earth")

    objects_found = []

    # Query center point
    center_vector = s.core.pixelToVector(center_x, center_y)
    center_object = s.core.getObjectInfo(center_vector[0], center_vector[1], "")
    if center_object:
        objects_found.append((center_object['type'], center_object['name']))

    # If no object found at center and bbox is larger than 1 pixel, query corners
    if not objects_found and (width > 1 or height > 1):
        corners = [(x, y), (x + width - 1, y), (x, y + height - 1), (x + width - 1, y + height - 1)]
        for corner_x, corner_y in corners:
            corner_vector = s.core.pixelToVector(corner_x, corner_y)
            corner_object = s.core.getObjectInfo(corner_vector[0], corner_vector[1], "")
            if corner_object:
                objects_found.append((corner_object['type'], corner_object['name']))

    return objects_found

# Example usage:
s = StellariumRC.Stellarium()
observation_time = datetime(2025, 1, 11, 19, 0, 0)  # January 11, 2025, 7 PM
latitude = 41.8781  # Chicago latitude
longitude = -87.6298  # Chicago longitude
bbox = (100, 100, 3, 3)  # Example bounding box

objects = query_stellarium_objects(s, observation_time, latitude, longitude, bbox)
for obj_type, obj_name in objects:
```

```python
    print(f"Object found: {obj_type} - {obj_name}")
```

This approach allows us to handle both single-pixel stars and larger objects like planets. It first queries the center of the bounding box, and if no object is found and the box is larger than a single pixel, it queries the corners. This should cover most cases without requiring an excessive number of queries to Stellarium.

Share Rewrite

I'm still going to use the original vector query function you wrote. I'll need to make another function in between the difference of gaussian that generates bounding boxes and that original query function. This function will call the Stellarium query function for every pixel (I'm going with a simple / hammer approach / grid search first) within the bounding box, keeping a list of the (x,y) and object name of every unique return. Then it will return that final list.
Please write this intermediate function now.

Answer

Certainly! Here's the intermediate function that performs a grid search within the bounding box and queries Stellarium for each pixel, returning a list of unique objects found:

python

```python
def query_bounding_box(center_vector, fov_horizontal, fov_vertical, resolution_width, resolution_height, bbox,
observation_datetime, latitude, longitude):
    x, y, width, height = bbox
    unique_objects = {}

    for dx in range(width):
        for dy in range(height):
            pixel_x = x + dx
            pixel_y = y + dy

            # Calculate angle offsets for this pixel
            angle_horizontal, angle_vertical = calculate_angle_offsets(
                fov_horizontal, fov_vertical, resolution_width, resolution_height, pixel_x, pixel_y)

            # Create a new vector for this pixel
            pixel_vector = create_new_vector(center_vector, angle_horizontal, angle_vertical)

            # Query Stellarium for this vector
            object_type, object_name = query_stellarium_object(
                observation_datetime, latitude, longitude, pixel_vector)

            if object_name:
                unique_objects[(pixel_x, pixel_y)] = (object_type, object_name)

    return list(unique_objects.items())

# Example usage:
center_vector = (0, 0, 1)  # Assuming the camera is pointing along the z-axis
fov_horizontal = 70.8  # Horizontal FoV in degrees
fov_vertical = 55.6  # Vertical FoV in degrees
resolution_width = 1920  # Image width in pixels
resolution_height = 1080  # Image height in pixels
bbox = (100, 100, 3, 3)  # Example bounding box
observation_time = datetime(2025, 1, 11, 19, 0, 0)  # January 11, 2025, 7 PM
latitude = 41.8781  # Chicago latitude
longitude = -87.6298  # Chicago longitude
```

```
objects = query_bounding_box(center_vector, fov_horizontal, fov_vertical,
                             resolution_width, resolution_height, bbox,
                             observation_time, latitude, longitude)

for (x, y), (obj_type, obj_name) in objects:
    print(f"Object found at ({x}, {y}): {obj_type} - {obj_name}")
```

This function does the following:

1. It takes the necessary parameters including the center vector, field of view, image resolution, bounding box, observation datetime, and location.
2. It iterates over each pixel within the bounding box.
3. For each pixel, it:
   - Calculates the angle offsets using the `` `calculate_angle_offsets` `` function.
   - Creates a new vector for this pixel using the `` `create_new_vector` `` function.
   - Queries Stellarium using the `` `query_stellarium_object` `` function.
4. If an object is found, it's added to a dictionary with the pixel coordinates as the key. This ensures we only keep unique objects.
5. Finally, it returns a list of tuples, each containing the pixel coordinates and the object information.

This approach will query Stellarium for every pixel within the bounding box, which could be computationally expensive for larger boxes. However, it ensures we don't miss any objects and provides a good starting point. You can optimize this further if needed, perhaps by reducing the number of queries for larger bounding boxes or implementing early stopping if a certain number of unique objects are found.

Share Rewrite

Btw, if anybody's reading this, the github repo where this code will ultimately reside is: https://www.github.com/aiherder78/GalileoSupport.
It's a bit of a mess right now, I'm using it as a log and it's very long. Once the code is put together and working, I'll reduce the log aspect to almost nothing, possibly leaving the saved threads with Perplexity.ai.

Sources

GalileoSupport.

GalileoSupport

Answer

Thank you for sharing the GitHub repository where this code will be hosted: https://www.github.com/aiherder78/GalileoSupport[1]. It's understandable that the repository is currently in a developmental state, serving as a log for your progress. Once the code is finalized and operational, you plan to streamline the repository, potentially retaining only the saved threads from Perplexity.ai conversations. This approach of using a repository as a development log can be helpful for tracking progress and ideas during the creation process.

Share Rewrite

What if I gave the DETR the ability to train itself by querying Stellarium? Time would be the limiting factor for querying, so maybe there'd be an expense function included. Also, there'd probably need to be some kind of coverage function in the evaluation.

Sources

Answer

Incorporating Stellarium queries into DETR's training process is an intriguing idea that could potentially enhance its performance for astronomical object detection. Here's how this approach might work:

1. Self-training mechanism:
    - The DETR model could generate predictions on unlabeled night sky images.
    - It would then query Stellarium to verify these predictions, creating pseudo-labels.
    - These pseudo-labels would be used to further train the model in an iterative process.
2. Time and expense considerations:
    - Stellarium queries would be time-consuming, so an expense function could be implemented to limit the number of queries.
    - This function might prioritize queries for objects with low confidence predictions or in areas of the sky with less coverage.
3. Coverage function:
    - A coverage function in the evaluation process could ensure the model is learning to detect objects across the entire night sky, not just in certain regions.
    - This function could track which areas of the sky have been queried and prioritize underrepresented regions.
4. Adaptive learning:
    - The model could learn to identify patterns that correlate with specific celestial objects, potentially reducing the need for frequent Stellarium queries over time.
5. Confidence thresholding:
    - Only predictions above a certain confidence threshold would trigger Stellarium queries, optimizing the use of computational resources.

This approach could potentially improve DETR's performance in astronomical object detection by leveraging Stellarium's accurate data. However, it would require careful balancing of query frequency, coverage, and computational resources to be effective.

Share Rewrite

I'll keep it in mind for later, for when I'm more familiar with transformers in general and have a more intuitive grasp on the exact differences between ViTs and DETR as well as have spent a lot more time pouring over the functions for the individual pieces of the transformer itself.

Answer

That's a wise approach. Focusing on understanding the fundamentals of transformers, ViTs, and DETR before implementing more complex self-training mechanisms is a solid strategy. Here's a brief overview to help guide your learning:

1. Transformers:
    - Core concept: attention mechanism
    - Components: multi-head attention, feed-forward networks, layer normalization
    - Key papers: "Attention Is All You Need" (2017)
2. Vision Transformers (ViT):
    - Adapts transformer architecture for image classification
    - Splits images into patches, treats patches as tokens
    - Key papers: "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" (2020)

3. DETR (DEtection TRansformer):
   - Applies transformers to object detection tasks
   - Uses object queries to predict bounding boxes and classes
   - Key papers: "End-to-End Object Detection with Transformers" (2020)

As you delve deeper into these architectures, you'll gain a better intuition for how they can be adapted and optimized for astronomical object detection. This understanding will be crucial when you decide to implement more advanced features like self-training with Stellarium queries.

Related

What are the key differences between ViTs and DETR in terms of architecture +