Documentation of AI training :

Step-by-Step Training Process

1. Image Data Preparation and Preprocessing

- **Image Loading**: Images are loaded from the filesystem using libraries like OpenCV (`cv2`). The images are read in a format like RGB or grayscale.
- **Normalization**: Pixel values of the images are scaled to a range [0, 1] or [-1, 1], which helps in stabilizing and speeding up the training process.
- **Resizing**: Images and masks are resized to a uniform size. This ensures that all images have the same dimensions, which is a requirement for batch processing in neural networks.

2. Data Augmentation

Albumentations: This library is used for applying various augmentations to the images and masks to increase the diversity of the training data. In your code, some specific augmentations include:

- **HorizontalFlip**: Flips the image horizontally, helping the model learn invariant features with respect to horizontal orientation.
- **RandomRotate90**: Rotates the image by 90 degrees randomly, providing rotational invariance.
- **ShiftScaleRotate**: Randomly shifts, scales, and rotates the image, which aids the model in becoming robust to changes in position, size, and orientation.
- **ColorJitter**: Randomly changes the brightness, contrast, saturation, and hue of the image, helping the model to generalize across varying lighting conditions.

3. Dataset and DataLoader Creation

Custom Dataset Class: A custom class inheriting from `torch.utils.data.Dataset` is created. This class handles:

- **Loading images and masks from the filesystem**: Images and corresponding masks are read from the file system.
- **Applying transformations (both augmentations and preprocessing)**: The augmentations mentioned above are applied to the data, along with any necessary preprocessing steps such as normalization and resizing.

- **Returning data in the form of tensors suitable for PyTorch models**: The transformed images and masks are converted into tensors which are then fed into the neural network.

DataLoader: PyTorch's `DataLoader` is used to:

- **Load the data in batches**: Splits the dataset into manageable batches for efficient processing.
- **Shuffle the data**: Randomizes the order of the data to prevent overfitting.
- **Handle parallel data loading**: Utilizes multiple worker processes to load the data, speeding up the training process.

4. Model Selection

Segmentation Model: In your code, the model used is **DeepLabV3+** from the `segmentation-models-pytorch` library. This model is designed specifically for segmentation tasks.

- **Backbone Encoder**: The backbone encoder used is **ResNet101**. This acts as the feature extractor, leveraging a pre-trained network on ImageNet to provide rich feature representations that aid in segmentation.

5. Loss Functions

In your code, the following loss functions are used:

- **Cross-Entropy Loss**: This is used for multi-class segmentation problems. It calculates the difference between the predicted probability distribution and the true distribution.
  - **Formula**: Loss= -$\sum$y.Log($\hat{y}$)
  - **Usage**: Suitable for cases where each pixel belongs to one of several classes.
- **Dice Loss**: This measures the overlap between the predicted segmentation and the ground truth segmentation. It is particularly useful for imbalanced datasets.
  - **Formula**: Loss=1- $\frac{2|A \cap B|}{|A|+|B|}$
  - **Usage**: Helps in improving the segmentation of smaller objects.
- **IoU (Intersection over Union) Loss**: Another measure of overlap between predicted and true segmentations.
  - **Formula**: IoU= $\frac{|A \cap B|}{|A \cup B|}$
  - **Usage**: Common in segmentation tasks to measure accuracy.

6. Optimizer

Adam: The optimizer used in your code is Adam (Adaptive Moment Estimation). It adjusts the learning rate based on the first and second moments of the gradients.

- **Formula**: $m_t=\beta_1 m_{t-1} + (1-\beta_1)g_t$

- **Usage**: Effective for most deep learning tasks due to its adaptive learning rate.

7. Training Loop

The training loop involves several key steps:

- **Epochs and Iterations**: Define the number of epochs (complete passes through the training dataset) and iterations (batches processed per epoch).
- **Forward Pass**: Input images are passed through the model to obtain predictions.
- **Loss Calculation**: Compute the loss using the defined loss functions (Cross-Entropy Loss, Dice Loss, IoU Loss). This measures how well the model's predictions match the ground truth.
- **Backward Pass**: Compute gradients using backpropagation. This involves calculating the gradient of the loss with respect to each model parameter.
- **Optimization**: Update model weights using the optimizer (Adam). This step adjusts the model parameters to minimize the loss.
- **Logging**: Track training progress by logging metrics like loss and accuracy. This helps in monitoring the training process and making necessary adjustments.

8. Validation

- **Validation Loop**: Periodically evaluate the model on a separate validation set.
- **Metrics Calculation**: Compute metrics such as accuracy, IoU, and Dice coefficient to monitor the model’s performance.

9. Checkpointing

- **Model Saving**: Save the model's state_dict (weights) periodically and when the validation performance improves.
- **Include Optimizer State**: Save the state of the optimizer to resume training seamlessly.

10. Model Evaluation

- **Comprehensive Testing**: Evaluate the model on a test set to ensure it generalizes well.
- **Visualization**: Visualize predictions to qualitatively assess the model’s performance.

11. Model Saving

- **Final Model Save**: Save the trained model to disk in a format like `.pt` or `.pth` for future use.
- **Include Metadata**: Ensure the saved model includes relevant metadata, like model architecture and preprocessing steps.

Dataset Used for Training

We have used open-source data for demo purposes which gives a good accuracy level, we can train our AI on any data set which minimal changes and preprocessing

Massachusetts Buildings Dataset

About Dataset

Context

Building Segmentation from Aerial Imagery is a challenging task. Obstruction from nearby trees, shadows of adjacent buildings, varying texture and color of rooftops, and varying shapes and dimensions of buildings are among other challenges that hinder present-day models in segmenting sharp building boundaries. High-quality aerial imagery datasets facilitate comparisons of existing methods and lead to increased interest in aerial imagery applications in the machine learning and computer vision communities.

Content

The Massachusetts Buildings Dataset consists of 151 aerial images of the Boston area, with each of the images being 1500 × 1500 pixels for an area of 2.25 square kilometers. Hence, the entire dataset covers roughly 340 square kilometers. The data is split into a training set of 137 images, a test set of 10 images, and a validation set of 4 images. The target maps were obtained by rasterizing building footprints obtained from the OpenStreetMap project. The data was restricted to regions with an average omission noise level of roughly 5% or less. A large amount of high-quality building footprint data was possible to collect because the City of Boston contributed building footprints for the entire city to the OpenStreetMap project. The dataset covers mostly urban and suburban areas and buildings of all sizes, including individual houses and garages, are included in the labels. The datasets make use of imagery released by the state of Massachusetts. All imagery is rescaled to a resolution of 1 pixel per square meter. The target maps for the dataset were generated using data from the OpenStreetMap project. Target maps for the test and validation portions of the dataset were hand-corrected to make the evaluations more accurate.

Acknowledgements

This dataset is derived from Volodymyr Mnih's original Massachusetts Buildings Dataset. Massachusetts Roads Dataset & Massachusetts Buildings dataset were introduced in Chapter 6 of his PhD thesis. If you use this dataset for research purposes you should use the following citation in any resulting publications:

```
@phdthesis{MnihThesis,
author = {Volodymyr Mnih},
title = {Machine Learning for Aerial Image Labeling},
school = {University of Toronto},
year = {2013}
}
```

Documentation of Website and Process Flow :

Home Page

Header (Navigation Bar):
- **Home:** A dropdown menu item that takes users to the homepage.
- **About :** A dropdown menu item providing information about the website.
- **Northern East India:** A dropdown menu item with sub-links to detailed information and Wikipedia pages of the states and districts in North Eastern India.

Main Content:
- Get Started Button: Prominently displayed on the homepage, leading users to the main functionality of the site.

Get Started Page

Main Content:
- **Search Button:** Allows users to enter a location.
- **Location Input Field:** Converts the entered location (string format) to latitudes and longitudes.

Functionality

1. Location Conversion:
  - The entered location string is converted to latitude and longitude coordinates.

2. Image Rendering:
  - The latitudes and longitudes render a 4km x 4km image of the region with a resolution of 1536 x 1536 pixels (depending on the availability of the data) per image, from the stored database using the GeoIndexing functionality of SQL.

3. Image Coordinates:
- The latitudes and longitudes of the centroid of the rendered images will be stored in a JSON file and passed for further processing.

Execution Flow Process of Pre-trained AI Model on Website:

1. **Setup and Preparation**
    - **Folder Structure**: Ensure the folder contains images and a JSON file with the center coordinates for each image.
    - **Pre-trained AI Model**: Load the pre-trained AI model that will be used to predict the masks for the images.
2. **Load Center Coordinates**
    - Read the JSON file to extract the center coordinates (latitude and longitude) for each image.
    - Store these coordinates in a dictionary or list for easy access during processing.
3. **Image Processing and Mask Prediction**
    - Iterate through each image in the folder.
    - For each image:
        - Load the image using an image processing library like OpenCV.
        - Pass the image through the pre-trained AI model to generate a mask.
        - The mask should highlight the building footprints as white patches on a black background.
4. **Binary Thresholding and Contour Detection**
    - Convert the predicted mask to a binary image using a thresholding technique.
    - Detects the contours in the binary image. These contours represent the boundaries of the white patches (building footprints).
5. **Convert Contours to Polygons**
    - For each detected contour:
        - Ensure the contour has enough points to form a valid polygon (usually at least 4 points).
        - Calculate the centroid of the contour using moments.
        - Convert the pixel coordinates of the contour and centroid to geographical coordinates using the image's center latitude and longitude.
6. **Pixel to Geographical Coordinates Conversion**
    - Define a function to convert pixel coordinates to geographical coordinates.
    - This conversion takes into account the image's center coordinates, the dimensions of the image, and an assumed pixel size (e.g., 1 meter per pixel).
7. **Prepare GeoJSON Features**
    - For each valid polygon:
        - Create a GeoJSON feature with the polygon's geographical coordinates.
        - Add properties such as the area and centroid in geographical coordinates.
8. **Compile GeoJSON File**
    - Collect all the GeoJSON features into a single FeatureCollection.
    - Save the FeatureCollection to a GeoJSON file.

Detailed Functions and Code Explanation

1. **Loading the Image and Predicting the Mask**
    - Use OpenCV to read each image.
    - Pass the image to the pre-trained AI model to get the predicted mask.
2. **Thresholding and Contour Detection**
    - Use `cv2.threshold` to convert the mask to a binary image.
    - Use `cv2.findContours` to detect the contours in the binary image.
3. **Calculating the Centroid**
    - Calculate the centroid of a contour using the moments calculated by `cv2.moments`.
4. **Coordinate Conversion Function**
    - Convert pixel coordinates to geographical coordinates using the image center and pixel size.
5. **Creating GeoJSON Features**
    - Create polygons from the contours.
    - Convert the pixel coordinates of the polygons and centroids to geographical coordinates.
    - Create GeoJSON features with the polygon coordinates and properties (area, centroid).
6. **Saving the GeoJSON File**
    - Compile all features into a FeatureCollection.
    - Save the FeatureCollection to a GeoJSON file using the `json` module.

Structure of the output:Explanation

4. Building Detection:

- A Computer Vision library forms a WKT (Well-Known Text) format polygon representing buildings and provides the coordinates of the centroid and the area of the building in a GeoJSON file in the following format:

```
json
feature = {
  "type": "Feature",
  "geometry": {
    "type": "Polygon",
    "coordinates": [polygon_coords]
  },
  "properties": {
    "area": area,
    "centroid": centroid_geo
  }
```

```
}
```

Retrieving and Preprocessing Solar data on website

Understanding Solar Radiation

Solar radiation is the energy emitted by the sun in the form of electromagnetic waves. This energy is crucial for various natural processes on Earth and is harnessed for generating solar power. Solar radiation can be divided into three main components:

1. **Global Horizontal Irradiance (GHI)**:
    - **Definition**: GHI is the total amount of shortwave radiation received from above by a surface horizontal to the ground. It includes both direct sunlight and diffuse sky radiation.
    - **Significance**: It is a key parameter for evaluating the potential of solar panels, as it represents the total solar power available per unit area.
2. **Direct Normal Irradiance (DNI)**:
    - **Definition**: DNI is the amount of solar radiation received per unit area by a surface that is always held perpendicular to the sun's rays.
    - **Significance**: This measure is essential for concentrating solar power (CSP) systems that focus sunlight onto a small area to generate heat or electricity.
3. **Diffuse Horizontal Irradiance (DHI)**:
    - **Definition**: DHI is the solar radiation received by a horizontal surface that molecules and particles have scattered in the atmosphere.
    - **Significance**: This component is crucial for understanding the total solar resource in areas with frequent cloud cover or atmospheric particulates.

Factors Affecting Solar Radiation

Several factors influence the amount and intensity of solar radiation received at a specific location:

1. **Geographical Location**: Latitude and altitude affect the angle and intensity of sunlight.
2. **Time of Year**: Seasonal variations due to the tilt of the Earth's axis influence the duration and intensity of sunlight.
3. **Time of Day**: The sun's position in the sky changes throughout the day, impacting solar radiation levels.
4. **Weather Conditions**: Cloud cover, humidity, and atmospheric particles can scatter and absorb sunlight, reducing the amount of solar radiation that reaches the surface.
5. **Surface Orientation**: The tilt and orientation of surfaces (such as solar panels) affect how much sunlight they receive.

Understanding these components and factors is essential for optimizing solar energy systems, improving energy forecasts, and conducting climate research (NSRDB).

Comprehensive Approach to Extracting, Analyzing, and Visualizing Solar Data for a Specified Location

Step-by-Step Process:

- **User Input and Geocoding**:
  - **Step**: The user provides the name of a location.
  - **Action**: Convert this location name to geographical coordinates (latitude and longitude) using a geocoding service Shapely is a Python package for set-theoretic analysis and manipulation of planar features using functions from the well-known and widely deployed GEOS library.
- **Fetching Solar Data**:
  - **Source**:NSRDB(National Solar Radiation Database)
  - **Details**: The dataset of the NSRDB has the following features:-
    - Resolution: 4 km x 4 km
    - Time Interval: 60 minutes
  - **Method:** API allows fetching solar radiation data for a given point for one year in CSV format.
  - **Limitation:**
    - Provide data on a particular point
    - API has a limitation of 5000 calls per day
    - Data is limited for span of 1 year
    - The interval is 60 min.
- **Data Segmentation**:
  - **Step**: Divide the data into four three-month periods:
    - GHI1: January to March
    - GHI2: April to June
    - GHI3: July to September
    - GHI4: October to December
- **Average Calculation**:
  - **Step**: Calculate the average solar radiation for each time group within each three months.
  - Now use solar energy production formula for each GHI1, GHI2 , GHI3 and GHI4 and display

5. Solar Data Matching:

- The latitudes and longitudes are matched with solar data provided by the government to calculate the average Global Horizontal Irradiance (GHI) for all seasons throughout the day.

6. Energy Potential Calculation:

- The energy potential for each building is calculated using the area, efficiency of the solar devices, respective GHI data, and daily sunlight duration. This information will be shown on the building footprints when the user hovers the building footprint on the map.

Formula = Area x GHI x Efficiency x SunlightDuration

javascript

const area = feature.properties.area;
const energyPotential_summer = area * ghi1 * efficiency * sunlight_duration;
const energyPotential_winter = area * ghi2 * efficiency * sunlight_duration;
const energyPotential_spring = area * ghi3 * efficiency * sunlight_duration;
const energyPotential_autumn = area * ghi4 * efficiency * sunlight_duration;

7. Map Interaction:

- When users hover over the map, a box displays the calculated energy potentials for the hovered region.

User Experience

Navigation:

- Users can easily navigate through the homepage, about page, and Northern East India page using the dropdown menus.

Interactive Map:

- Users can interact with the map to view detailed energy potential information for specific buildings, enhancing the user experience with real-time data.

- Users can Zoom in and out, toggle over neighboring areas, and search new area details.

- Users can choose which layer to be shown on the screen, from the available layers such as, Map, satellite imagery, Building Footprint, etc.

This website provides a comprehensive tool for users to analyze the solar energy potential of buildings in a specified region, using advanced image processing and data analysis techniques.

Proposed Architecture

1. Frontend
Technologies:
- HTML, CSS, JavaScript
- React.js for building user interfaces
- Leaflet.js for interactive maps

Pages:

Home Page
Header (Navigation Bar):
- **Home:** Dropdown menu item that takes users to the homepage.
- **About:** Dropdown menu item providing information about the website.
- **Northern East India:** Dropdown menu item with sub-links to detailed information about the states in North Eastern India.

Main Content:
- **Get Started Button:** Prominently displayed on the homepage, leading users to the main functionality of the site.

Get Started Page
Main Content:
- **Search Button:** Allows users to enter a location.
- **Location Input Field:** Converts the entered location (string format) to latitudes and longitudes.

2. Backend

Technologies:
- Node.js with Express for server-side operations
- Python for image processing and data analysis
- PostgreSQL/PostGIS for spatial data storage and queries

API Endpoints:
- **/api/location:** Converts location string to latitude and longitude.
- **/api/image:** Generates and preprocesses images.
- **/api/buildings:** Detects buildings and generates GeoJSON data.
- **/api/solar:** Matches solar data and calculates energy potential.
- **/api/map:** Provides interactive map data.

3. Database

Database Schema:

Locations Table:
- `id` (Primary Key)
- `name` (String)
- `latitude` (Float)
- `longitude` (Float)

Buildings Table:
- `id` (Primary Key)
- `location_id` (Foreign Key to Locations)
- `polygon` (Polygon in WKT)
- `centroid` (Point in WKT)
- `area` (Float)

SolarData Table:
- `id` (Primary Key)
- `latitude` (Float)
- `longitude` (Float)
- `ghi_summer` (Float)
- `ghi_winter` (Float)
- `ghi_spring` (Float)
- `ghi_autumn` (Float)

4. Image Processing

Pipeline:
1. **Location Conversion:**
   - Convert location string to latitude and longitude using a geocoding API (e.g., Google Maps API).

2. **Image Rendering:**
   - Use latitude and longitude to render a 4km x 4km image using a mapping service (e.g., Mapbox, Google Maps).

3. **Image Preprocessing:**
   - Convert the image to black-and-white, highlighting buildings using OpenCV or similar libraries.

4. **Building Detection:**
   - Detect buildings and create WKT polygons using a computer vision library (e.g., OpenCV, TensorFlow).
   - Calculate centroid and area, store in GeoJSON format.

5. Solar Data Matching

- Use the latitude and longitude to retrieve GHI data from government datasets or APIs.
- Calculate average GHI for all seasons.

6. Energy Potential Calculation

Formula:
```javascript
const area = feature.properties.area;
const energyPotential_summer = area * ghi1 * efficiency * sunlight_duration;
const energyPotential_winter = area * ghi2 * efficiency * sunlight_duration;
const energyPotential_spring = area * ghi3 * efficiency * sunlight_duration;
const energyPotential_autumn = area * ghi4 * efficiency * sunlight_duration;
```

7. Interactive Map

- Use Leaflet.js to create an interactive map.
- Display energy potential information when users hover over buildings.
- Fetch and display real-time data using backend API.

User Experience

Navigation:
- Intuitive dropdown menus for easy navigation through the site.

Interactive Map:
- Real-time data display for enhanced user interaction and engagement.

This architecture ensures a robust and scalable platform for analyzing solar energy potential, leveraging modern web technologies, and advanced data processing techniques.