

**REPORT OF IMAGE CAPTIONING AND  
SEGMENTATION MODEL  
Vision AI Suite**

**ASSIGNED BY  
ZIDIO DEVELOPMENT**



**GUIDED BY -  
CHANDAN MISHRA SIR**

## **PROJECT OVERVIEW**

This project focuses on the dual task of Image Captioning and Image Segmentation. Image Captioning involves generating descriptive textual captions for a given image using deep learning models, while Image Segmentation involves identifying and labeling regions of an image with corresponding objects or categories. Combining these two tasks will give interns hands-on experience with computer vision, natural language processing, and deep learning concepts.

1. Understand and implement state- of-the-art models for image captioning.
2. Train and evaluate models on standard datasets (e.g., MS COCO, Pascal VOC).
3. Perform semantic and instance segmentation on images.
4. Perform semantic and instance segmentation on images.
5. Improve accuracy and performance of both tasks through experimentation.

## **TECHSTACK REQUIRED**

- ◆ Python
- ◆ TensorFlow / PyTorch
- ◆ OpenCV
- ◆ NLTK / spaCy (for language preprocessing)
- ◆ Flask / Streamlit (for deployment)
- ◆ Jupyter Notebook

# **WORKFLOW**

- 1. Understanding the Concepts and Literature Review**
- 2. Dataset collection.**
- 3. Environment Setup.**
- 4. Dividing the Project into two parts.**
  - I. Coping up with the Image Captioning Model.**
    - Dataset Preparation.
    - Data Preprocessing.
    - Modelling the Architecture:
      - Encoder.
      - Decoder.
    - Creating the training script.
    - Evaluation with BLEU, METEOR Matrices.
    - Fine Tuning the Model.
    - Deploying the webpage on streamlit local host.
  - II. Proceeding with Segmentation.**
    - Data Preprocessing.
    - Modelling the Architecture of U-net.
    - Creating the training script.
    - Deploying the webpage on streamlit local host.
- 5. Pipelining:**
  - Integrating Captioning Model. (Trained on default Image-net Weights)
  - Mask R-CNN and Faster R-CNN for Object and Instance detection (Pretrained Models).
  - U-net model. (Trained on default Image-net Weights)

# **ACKNOWLEDGEMENT**

**I would like to express my deepest gratitude to those who have guided and supported me throughout this project's development.**

**First, I wish to thank ZIDIO DEVELOPMENTS , for giving me the chance and for all the support I have received .**

**I would like to express my highest gratitude to CHANDAN MISHRA SIR , for the efforts given by him to help me successfully complete the project on the given deadline.**

**Finally, I would like to be to use this great opportunity to extend my heartfelt acknowledgment to every other person that in one way or the other contributed to the success of this project.**

**Yours Sincerely,  
Prottus Manna.**

# INTRODUCTION

In the rapidly evolving fields of computer vision and artificial intelligence, the ability to interpret and describe visual data is of paramount importance. The dual tasks of image captioning and image segmentation have emerged as foundational techniques, enabling machines to not only generate meaningful textual descriptions of images but also to accurately identify and delineate objects within them. This project, “Image Captioning and Segmentation,” seeks to bridge these two domains by developing an integrated system capable of both generating descriptive captions for images and performing precise segmentation of visual elements.

Image captioning leverages deep learning models to produce coherent and contextually relevant textual summaries of visual content, facilitating enhanced human-computer interaction and accessibility. Meanwhile, image segmentation assigns semantic or instance-based labels to every pixel in an image, allowing for detailed understanding and analysis of complex scenes. By combining these capabilities, the project offers a comprehensive approach to visual understanding, with applications ranging from autonomous vehicles and medical imaging to digital content creation and assistive technologies.

# 1. PROCEEDING WITH THE LITERATURE REVIEW

- PMC Article

This review looks at the development of image captioning. It starts with early statistical models and moves to today's deep learning methods. It covers how computer vision and natural language processing work together. It explains attention mechanisms, including soft, hard, and adaptive attention, and compares their performance. The paper also points out well-known datasets and evaluation metrics. This makes it a good starting point for understanding basic concepts and the latest methods in image captioning.

- Gain Coding Base Knowledge Source: Papers with Code

This resource collects recent research papers, code implementations, and leaderboards for image captioning tasks. It is very useful for keeping up with the latest models, datasets, and evaluation standards. You can also find open-source code to experiment with or build on.

- Videos

I. <https://youtu.be/lhsX8ISASus?si=POMMu-04vXVK6sv->

II. [https://youtube.com/playlist?list=PL12YwfULs0pnL\\_9Pj6udM6PE2-SWZbTIX&si=MLCTi63xR9b2\\_JKK](https://youtube.com/playlist?list=PL12YwfULs0pnL_9Pj6udM6PE2-SWZbTIX&si=MLCTi63xR9b2_JKK)

III. <https://youtu.be/lhq1t7NxS8k?si=Zr8udz57uiDUsxT4>

IV. <https://youtu.be/WgPbbWmnXJ8?si=3S2NYP6CIKJGYVPS>

## 2. STARTING WITH CHOOSING DATA SET AND DOWNLOADING

I chose to work with MS COCO 2017 Dataset because according to me it was a descent dataset to train and build an image captioning and segmentation from scratch and also without using any-kind of pretrained models.

G

Link to MS COCO 2017 - <https://cocodataset.org/#download>

**Downloaded the Following files :**

- **annotations\_trainval2017.zip**
- **train2017.zip**
- **val2017.zip**

Next step was to extract the files to the project directory that I had created.

After downloading the files I extracted it to the specified folder. Created a properly accessible and a totally separate directory for the project with the name – `image_captioning_segmentation`.



### 3. PROCEEDING TO ENVIRONMENT SETUP STAGE

**STEP 1 -- Changing the directory to my working directory.**

**STEP 2 -- Creating a venv environment inside the directory.**

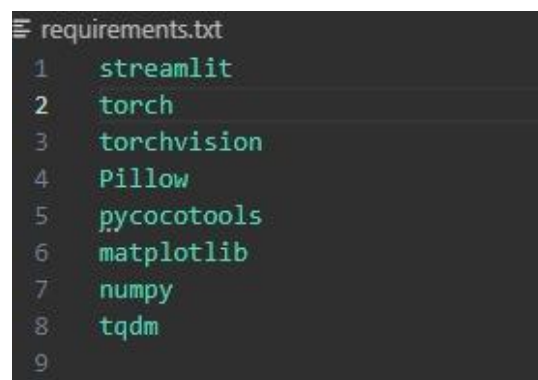
**Command - `python -m venv venv`**

**STEP 3 -- Activating the venv environment .**

**Command – `venv\Scripts\activate`**

**STEP 4 -- Now we need to install all the requirements in within the activated venv environment.**

**REQUIREMENTS FOLDER:**



```
requirements.txt
1  streamlit
2  torch
3  torchvision
4  Pillow
5  pycocotools
6  matplotlib
7  numpy
8  tqdm
9
```

**Spacy was also installed separately with the command:**

**`Python -m spacy download en_core_web_sm`**

**STEP 5 - Verifying if all the packages are installed properly or not.**

## ERRORS FACED WHILE SETTING UP THE ENVIRONMENT

```
(caption-env) C:\Users\Pratyush>import nltk
'import' is not recognized as an internal or external command,
operable program or batch file.

(caption-env) C:\Users\Pratyush>nltk.download('punkt')
'nltk.download' is not recognized as an internal or external command,
operable program or batch file.
```

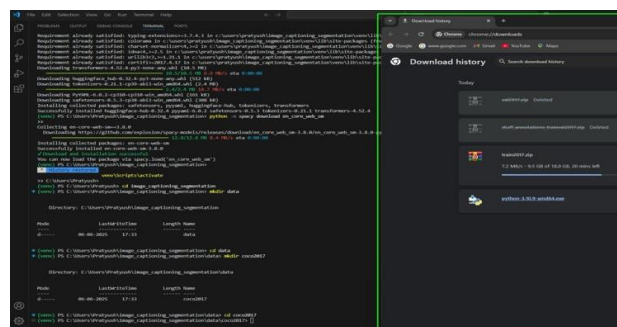
```
caption-env) C:\Users\Pratyush>pip install tensorflow
ERROR: Could not find a version that satisfies the requirement tensorflow (from versions: none)
ERROR: No matching distribution found for tensorflow

caption-env) C:\Users\Pratyush>
```

```
PS C:\Users\Pratyush> cd image_captioning_segmentation
>> python -m venv venv
>>
Python was not found; run without arguments to install from the Microsoft Store, or disable this shortcut from Settings > Apps > Advanced app settings > App execution aliases.
PS C:\Users\Pratyush> image_captioning_segmentation> |
```

```
PS C:\Users\Pratyush\image_captioning_segmentation> venv\Scripts\activate
>>
venv\Scripts\activate : File C:\Users\Pratyush\image_captioning_segmentation\venv\Scripts\Activate.ps1 cannot be loaded because running scripts is disabled on this system. For more information, see about_Execution_Policies at
https://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ ~~~~~
+ ~~~~~
+ CategoryInfo          : SecurityException ( ( ) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
PS C:\Users\Pratyush\image_captioning_segmentation>
```

**FINALLY RESOLVED ALL THE ERRORS AND  
INSTALLED ALL THE REQUIREMENTS AND  
SUCCESSFULLY COMPLETED THE FIRST STEP.**



## 4. I. STARTING WITH THE IMAGE CAPTIONING

### ● DATASET PREPARATION

Before proceeding with the data preprocessing, First we need to prepare the dataset properly so that it can be processed fluently without any issues. We can say dataset preparation is the most important step that needs to be performed after downloading the data from internet.

## CODE BREAKDOWN AND PROPER LEARNING EXPLANATION

### 1. Imports

```
import torch
from torch.utils.data import Dataset
from PIL import Image
import os
import nltk
```

- **torch:** The main PyTorch library for deep learning.
- **Dataset:** Base class from PyTorch for custom datasets.
- **PIL.Image:** For opening and processing images.
- **os:** For handling file paths.
- **nltk:** For tokenizing captions.

## 2. Class Definition

```
class CocoCaptionDataset(Dataset):
```

- Defines a custom dataset class inheriting from PyTorch's Dataset.
- This allows integration with PyTorch's data loading utilities.

## 3. Initialization ( \_\_init\_\_ )

```
def __init__(self, image_folder, annotations, word2idx, transform=None):  
    self.image_folder = image_folder  
    self.annotations = annotations  
    self.word2idx = word2idx  
    self.transform = transform
```

- image\_folder: Path to the folder containing images.
- annotations: List of caption annotations (each is a dict with keys like 'image\_id' and 'caption').
- word2idx: Dictionary mapping words to integer indices.
- transform: Optional image transformation (e.g., resizing, normalization).

## 4. Length Method ( \_\_len\_\_ )

```
def __len__(self):  
    return len(self.annotations)
```

- Returns the number of samples in the dataset (equal to the number of annotations).

## 5. Get Item Method ( \_\_getitem\_\_ )

```
def __getitem__(self, idx):
    ann = self.annotations[idx]
    image_id = ann['image_id']
    caption = ann['caption']
    image_filename = f"{image_id:012d}.jpg"
    image_path = os.path.join(self.image_folder, image_filename)
    image = Image.open(image_path).convert("RGB")
    if self.transform:
        image = self.transform(image)
    tokens = ['<start>'] + nltk.tokenize.word_tokenize(caption.lower()) + ['<end>']
    caption_indices = [self.word2idx.get(token, self.word2idx['<unk>']) for token in tokens]
    return image, torch.tensor(caption_indices)
```

- **def \_\_getitem\_\_(self, idx):**
  - Returns the idx-th sample from the dataset.
- **Inside \_\_getitem\_\_**
  - **ann = self.annotations[idx]** : Gets the annotation at index idx.
  - **image\_id = ann['image\_id']** : Extracts the image ID from the annotation.
  - **caption = ann['caption']** : Extracts the caption text.
  - **image\_filename = f"{image\_id:012d}.jpg"** : Formats the image filename with leading zeros (COCO standard: 12 digits).
  - **image\_path = os.path.join(self.image\_folder, image\_filename)** : Constructs the full path to the image file.
  - **image = Image.open(image\_path).convert("RGB")** : Opens the image and ensures it is in RGB format.
  - **if self.transform:**
    - **image = self.transform(image)** : Applies the specified image transformation (e.g., resizing, normalization) if provided.
  - **tokens = ['<start>'] + nltk.tokenize.word\_tokenize(caption.lower()) + ['<end>']** :
    - Tokenizes the caption into words (lowercase).
    - Adds special tokens <start> and <end> to mark the beginning and end of the caption.

- `caption_indices = [self.word2idx.get(token, self.word2idx['<unk>']) for token in tokens] :`
  - Converts each token to its corresponding index using `word2idx`.
  - If a token is not found, uses the index for `<unk>` (unknown token).
- `return image, torch.tensor(caption_indices) :` Returns the transformed image and the caption as a tensor of indices.

## ● DATASET PREPROCESSING

Before starting with the Dataset Preprocessing first I need to import the required packages like `json` , `nlTK` , `os` , also `counter`. Next, I had to download the `nlTK` tokenizer data.

Then the desired coding is done based on compatibility. In code everything is mention and why I am using each segment of the code is given properly. In the dataset preprocessing first the path is selected then the captions are loaded. After loading the captions they are tokenized. After that I worked with the creation of vocabulary with `word2idx`.

## CODE BREAKDOWN AND PROPER LEARNING EXPLANATION

### 1. Import Statements

```
import nltk
import os
from collections import Counter
```

- json: For reading JSON files (used for captions).
- nltk: The Toolkit, used for natural language processing tasks like tokenization.
- os: For handling file and directory paths.
- collections.Counter: For counting occurrences of words in the vocabulary.

## **2. NLTK Data Download**

```
nltk.download('punkt')  
nltk.download('punkt_tab')
```

- Downloads the punkt tokenizer models, which are required for sentence and word tokenization.
- punkt\_tab is not standard; likely intended to ensure compatibility or additional tokenization data, but usually punkt is sufficient.

## **3. Define Captions File Path**

```
CAPTIONS_PATH = os.path.join( "C:\\Users", "Pratyush",  
"image_captioning_segmentation", "data", "coco2017", "annotations_trainval2017",  
"annotations", "captions_train2017.json"  
)
```

- Constructs the path to the COCO captions JSON file.
- Uses os.path.join for cross-platform path compatibility.

## **4. Check File Existence**

```
if not os.path.isfile(CAPTIONS_PATH):
    raise FileNotFoundError(f"Could not find {CAPTIONS_PATH}. Please check your dataset setup.")
```

- Ensures the captions file exists before proceeding, preventing errors due to missing files.

## 5. Load Captions Data

```
with open(CAPTIONS_PATH, 'r', encoding='utf-8') as f:
    captions_data = json.load(f)
```

- Opens and reads the JSON file containing image captions.
- Stores the data in captions\_data.

## 6. Tokenize Captions

```
captions = []
for annot in captions_data.get('annotations', []):
    caption = annot.get('caption')
    if caption is not None:
        tokens = nltk.tokenize.word_tokenize(caption.lower())
        captions.append(tokens)
    else:
        print(f"Warning: annotation without 'caption' key: {annot}")
```

- Extracts each caption from the annotations.
- Converts each caption to lowercase and tokenizes it into words using nltk.tokenize.word\_tokenize.
- Stores the tokenized captions in the captions list.
- Prints a warning if a caption is missing.

## 7. Check Captions Presence

```
if not captions:
    raise ValueError("No captions found in the dataset.")
```



- Raises an error if no captions were found, ensuring the script does not proceed with empty data.

## 8. Create Vocabulary (Basic Version)

```
word_counts = Counter()
for tokens in captions:
    word_counts.update(tokens)
```

- Counts the occurrences of each word across all captions using Counter.

## 9. Remove Rare Words

```
threshold = 5
words = [word for word, count in word_counts.items() if count >= threshold]
print("Vocabulary size:", len(words))
```

- Filters out words that appear fewer than 5 times.
- Prints the size of the resulting vocabulary.

## 10. Import Pickle

```
import pickle
```

- Imports the pickle module for serializing Python objects to disk.

## 11. Add Special Tokens

```
special_tokens = ['<pad>', '<start>', '<end>', '<unk>']
words = special_tokens + [w for w in words if w not in special_tokens]
```

- Adds special tokens for padding, start-of-sequence, end-of-sequence, and unknown words.
- Ensures these tokens are included in the vocabulary.

## 12. Create Word-to-Index and Index-to-Word Mappings

```
word2idx = {word: idx for idx, word in enumerate(words)}  
idx2word = {idx: word for word, idx in word2idx.items()}
```

- Creates two dictionaries:
  - word2idx: Maps each word to a unique integer index.
  - idx2word: Maps each index back to its corresponding word.

## 13. Print Sample Mapping

```
print("word2idx sample:", list(word2idx.items())[:10])
```

- Prints the first 10 entries in word2idx for inspection.

## 14. Save Mappings to Disk

```
with open('word2idx.pkl', 'wb') as f:  
    pickle.dump(word2idx, f)  
with open('idx2word.pkl', 'wb') as f:  
    pickle.dump(idx2word, f)  
print("Vocabulary dictionaries saved to word2idx.pkl and idx2word.pkl")
```

- Saves the word2idx and idx2word dictionaries to disk using pickle.

- Prints a confirmation message.

## **Techniques and Modern Processes Used in My Script**

- Tokenization:
  - Uses NLTK's word tokenize for splitting sentences into words.
- Vocabulary Construction:
  - Builds a vocabulary from the most frequent words, filtering out rare ones to reduce noise and model size.
- Special Tokens:
  - Uses <pad>, <start>, <end>, and <unk> for sequence padding, marking the start and end of sequences, and handling unknown words—common in modern NLP pipelines.
- Serialization:
  - Uses pickle to save mappings for later use in model training.
- Error Handling:
  - Checks for file existence and data validity before proceeding.

### **Name of Script in Repo :**

● *data\_preprocessing.py*

- **MODELLING THE ARCHITECTURE**  
**CNN + LSTM BASED**
- **ENCODER (CNN)**

## 1. Import Statements

```
import torch
import torch.nn as nn
from torchvision.models import resnet50
```

- torch: Core PyTorch library
- torch.nn: Neural network modules
- torchvision.models.resnet50: Pretrained ResNet-50 model

## 2. Class Definition

```
class EncoderCNN(nn.Module):
```

- Defines a custom CNN encoder that inherits from PyTorch's base module class
- Core component for image feature extraction in captioning models

### 3. Initialization ( \_\_init\_\_ )

```
def __init__(self, embed_size):
    super(EncoderCNN, self).__init__()
    resnet = resnet50(weights="IMAGENET1K_V1")
    modules = list(resnet.children())[:-1]
    self.resnet = nn.Sequential(*modules)
    self.linear = nn.Linear(resnet.fc.in_features, embed_size)
    self.bn = nn.BatchNorm1d(embed_size, momentum=0.01)
```

- `super()`: Initializes parent `nn.Module` class
- `resnet = resnet50(...)`: Loads ResNet-50 with ImageNet-pretrained weights
- `modules = ...[:-1]`: Removes the final classification layer (keeps convolutional features)
- `self.resnet`: Creates feature extractor (all layers except last)
- `self.linear`: Projects features to embedding space
- `self.bn`: Batch normalization for stabilized training

### 4. Forward Pass (forward)

```
def forward(self, images):
    with torch.no_grad():
        features = self.resnet(images)
        features = features.squeeze(-1).squeeze(-1)
    features = self.linear(features)
    features = self.bn(features)
    return features
```

- `with torch.no_grad()`: Freezes ResNet weights (no gradient computation)
- `self.resnet(images)`: Extracts features (output shape: [batch, 2048, 1, 1])
- `squeeze()`: Removes spatial dimensions (result: [batch, 2048])

- `linear()`: Projects to embedding space [batch, embed\_size]
- `bn()`: Normalizes embeddings
- Returns final image embeddings

## Key Techniques Used

### 1. Transfer Learning:

- Uses pretrained ResNet-50 weights (IMAGENET1K\_V1)
- Freezes convolutional layers during training

### 2. Feature Extraction:

- Removes classification head to access spatial features
- Uses global average pooling implicitly via `squeeze()`

### 3. Dimensionality Reduction:

- Linear projection from 2048-dim features to custom `embed_size`
- Batch normalization for stable training

### 4. Computational Optimization:

- Gradient disabling for pretrained weights
- Sequential container for efficient processing

## • DECODER(LSTM)

### 1. Class Definition & Initialization

```
class DecoderRNN(nn.Module):
    def __init__(self, embed_size, hidden_size, vocab_size, num_layers=1):
        super(DecoderRNN, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size,
                             num_layers, batch_first=True)
        self.linear = nn.Linear(hidden_size, vocab_size)
        self.dropout = nn.Dropout(0.2)
```

- **nn.Embedding**: Converts word indices → dense vectors of size `embed_size`.
- **nn.LSTM**:
  - Input: `embed_size` (word vector size)
  - Hidden state: `hidden_size`
  - `num_layers`: Stacked LSTM layers (default=1)
  - `batch_first=True`: Input/Output shapes are (batch, seq\_len, features)
- **nn.Linear**: Maps LSTM output → vocabulary space for word prediction.
- **nn.Dropout**: Regularization (20% dropout) to prevent overfitting . Adjusted later, according to requirements.

## 2. Forward Pass

```
def forward(self, features, captions):
    embeddings = self.dropout(self.embed(captions)) # [batch_size, seq_len, embed_size]
    # Concatenate image features as the first input
    features = features.unsqueeze(1) # [batch_size, 1, embed_size]
    inputs = torch.cat((features, embeddings[:, :-1, :]), dim=1) # [batch_size, seq_len, embed_size]
    hidden, _ = self.lstm(inputs)
    outputs = self.linear(hidden) # [batch_size, seq_len, vocab_size]
    return outputs
```

- **Convert Caption Indices to Word Vectors** : `embeddings = self.dropout(self.embed(captions))` :
  - captions are integer indices representing words (shape: [batch, seq\_len]).
  - `self.embed(captions)` maps each index to a dense vector (embedding) of size `embed_size`.
  - `self.dropout(...)` applies dropout to these embeddings for regularization (helps prevent overfitting).
- **Prepare Initial Input** : `features = features.unsqueeze(1)` :
  - features are the image features from the encoder (shape: [batch, embed\_size]).
  - `unsqueeze(1)` adds a sequence-length dimension, making it [batch, 1, embed\_size].

This prepares the image features to be the first input to the LSTM, acting like a “start token” for the sequence.

- **Combine Image Features with Shifted Caption Embeddings** : inputs = `torch.cat((features, embeddings[:, :-1, :]), dim=1)` :
  - `embeddings[:, :-1, :]` takes all but the last word in each caption (shape: [batch, seq\_len-1, embed\_size]).
  - `torch.cat((features, ...), dim=1)` concatenates the image features with the shifted caption embeddings along the sequence dimension.

**Teacher forcing:** During training, the model is given the correct previous word at each step. The input at step  $t$  is the word at step  $t-1$  (or image features at step 0), so the model learns to predict the next word in the sequence.

- **Process Sequence Through LSTM** : `hiddens, _ = self.lstm(inputs)` :
  - Feeds the combined input sequence (image features + shifted captions) through the LSTM.
  - `hiddens`: Output hidden states for each timestep (shape: [batch, seq\_len, hidden\_size]).
  - `_`: Discards the final hidden/cell state (not needed here).
- **Predict Next Words** : `outputs = self.linear(hiddens) # [batch, seq_len, vocab_size]` **return** outputs.
  - `self.linear(hiddens)` projects each hidden state to the vocabulary size.

These scores are used to compute the loss during training (e.g., cross-entropy loss against the true next words).



# **Techniques Used**

## **1. Teacher Forcing :**

- During training: Uses *ground truth* previous words (from captions) as input.
- At inference: Uses *predicted* words autoregressively.

## **2. Sequence Shift :**

- Input: embeddings[:, :-1] (words 0 to n-1)
- Target: captions (words 1 to n)
- Aligns inputs with next-word prediction.

## **3. Image Feature Integration :**

- Image features seed the decoder as the first “word” in the sequence.

## **4. Regularization :**

- Dropout on embeddings prevents overfitting.

## • CREATING THE TRAINING SCRIPT

### 1. Setup and Device Configuration

```
import os
import torch
print("CUDA available:", torch.cuda.is_available())
if torch.cuda.is_available():
    print("GPU Name:", torch.cuda.get_device_name(0))
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

- Checks if CUDA (GPU) is available and prints the GPU name if so.
- Sets the device to use GPU if available, otherwise CPU.
- **Purpose:** Ensures the model runs on the fastest available hardware.

### 2. Image Transformations

```
from torchvision import transforms

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

- **Resize:** Adjusts all images to 224x224 pixels.
- **ToTensor:** Converts images to PyTorch tensors.
- **Normalize:** Normalizes images using mean and std from ImageNet.
- **Purpose:** Standardizes image input for the CNN encoder

### 3. Data Loading and Collation

```
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence

def collate_fn(batch):
    images, captions = zip(*batch)
    images = torch.stack(images)
    captions = pad_sequence(captions, batch_first=True, padding_value=0)
    return images, captions
```

- `collate_fn`: Custom function to combine images and captions into batches.
- `torch.stack(images)`: Stacks images into a tensor.
- `pad_sequence`: Pads captions to the same length in each batch (using <pad> token index 0).
- Purpose: Handles variable-length captions in each batch.

### 4. Data Loading and Splitting

```
import json
from scripts.dataset import CocoCaptionDataset

with
open(r'C:\Users\Pratyush\image_captioning_segmentation\data\coco2017\annotations_trainval2017\annotations\captions_train2017.json', 'r') as f:
    captions_data = json.load(f)
    annotations = captions_data['annotations']

import pickle
with open('word2idx.pkl', 'rb') as f:
    word2idx = pickle.load(f)
with open('idx2word.pkl', 'rb') as f:
    idx2word = pickle.load(f)
```

- Loads COCO captions annotations from a JSON file.
- Loads word-to-index and index-to-word mappings from pickle files.
- **Purpose**: Prepares data and vocab for model training

## 5. Train/Validation Split

```
from sklearn.model_selection import train_test_split
train_ann, val_ann = train_test_split(annotations, test_size=0.2,
random_state=42)
```

- Splits annotations into training (80%) and validation (20%) sets.
- **Purpose:** Enables model evaluation on unseen data.

## 6. Dataset and Data-Loader Creation

```
train_dataset = CocoCaptionDataset(
    image_folder='data/coco2017/train2017',
    annotations=train_ann,
    word2idx=word2idx,
    transform=transform
)
val_dataset = CocoCaptionDataset(
    image_folder='data/coco2017/train2017',
    annotations=val_ann,
    word2idx=word2idx,
    transform=transform
)
train_loader = DataLoader(train_dataset, batch_size=32,
shuffle=True, collate_fn=collate_fn)
val_loader = DataLoader(val_dataset, batch_size=32,
shuffle=False, collate_fn=collate_fn)
```

- Creates training and validation datasets.
- Uses DataLoader for efficient batch loading and shuffling.
- **Purpose:** Feeds data to the model in batches

## 7. Test Data Loading

```
for images, captions in train_loader:
    print("Images batch shape:", images.shape)
    print("Captions batch shape:", captions.shape)
    print("First caption indices:", captions[0])
    break # Only check the first batch
```

- Loads and prints the first batch to verify data shapes and contents.
- Purpose: Debugging and data sanity check.

## 8. Model Initialization

```
from models.encoder import EncoderCNN
from models.decoder import DecoderRNN

embed_size = 256
hidden_size = 512
vocab_size = len(word2idx)

encoder = EncoderCNN(embed_size).to(device)
decoder = DecoderRNN(embed_size, hidden_size, vocab_size).to(device)
```

- Creates encoder (CNN) and decoder (RNN) models.
- Moves models to the selected device (GPU/CPU).
- Purpose: Defines the image captioning model architecture

## 9. Loss, Optimizer, and Training Setup

```
import torch.nn as nn
import torch.optim as optim

criterion = nn.CrossEntropyLoss(ignore_index=word2idx['<pad>'])
params = list(decoder.parameters()) + list(encoder.linear.parameters()) +
list(encoder.bn.parameters())
optimizer = optim.Adam(params, lr=1e-3)

num_epochs = 58
patience = 8 # Early stopping patience
```

- criterion: Cross-entropy loss, ignoring padding tokens.
- optimizer: Adam optimizer for model parameters.
- num\_epochs: Number of training epochs.
- patience: Early stopping patience (epochs without improvement).
- Purpose: Configures loss, optimization, and training loop

## 10. Checkpoint Loading (Resume Training)

```
start_epoch = 0
best_val_loss = float('inf')
epochs_no_improve = 0

if os.path.exists('checkpoint.pth'):
    print("Loading checkpoint...")
    checkpoint = torch.load('checkpoint.pth', map_location=device)
    encoder.load_state_dict(checkpoint['encoder_state_dict'])
    decoder.load_state_dict(checkpoint['decoder_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    start_epoch = checkpoint['epoch'] + 1
    best_val_loss = checkpoint['best_val_loss']
    epochs_no_improve = checkpoint['epochs_no_improve']
    print(f"Resuming training from epoch {start_epoch}")
else:
    print("No checkpoint found, starting from scratch.")
```

- Loads model, optimizer, and training state from checkpoint if available.
- Purpose: Enables resuming interrupted training

## 11. Validation Evaluation Function

```
def evaluate_on_validation(encoder, decoder, val_loader, criterion, device):
    encoder.eval()
    decoder.eval()
    val_loss = 0
    with torch.no_grad():
        for images, captions in val_loader:
            images = images.to(device)
            captions = captions.to(device)
            features = encoder(images)
            outputs = decoder(features, captions[:, :-1])
            loss = criterion(outputs.reshape(-1, outputs.size(2)),
captions[:, 1:].reshape(-1))
            val_loss += loss.item()
    return val_loss / len(val_loader)
```

## 12. Training Loop

```
print("Starting training...")
for epoch in range(start_epoch, num_epochs):
    encoder.train()
    decoder.train()
    total_loss = 0
    for i, (images, captions) in enumerate(train_loader):
        images = images.to(device)
        captions = captions.to(device)
        features = encoder(images)
        outputs = decoder(features, captions[:, :-1])
        loss = criterion(outputs.reshape(-1, outputs.size(2)),
captions[:, 1:].reshape(-1))
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        if (i+1) % 20 == 0:
            print(f"Epoch [{epoch+1}/{num_epochs}], Step
[{i+1}/{len(train_loader)}], Loss: {loss.item():.4f}")
    avg_loss = total_loss / len(train_loader)
```

- Iterates over training data, computes loss, and updates model weights.
- Teacher Forcing: Uses ground truth captions as input to the decoder.
- Purpose: Trains the model to generate accurate captions

## 13. Validation and Early Stopping

```
avg_val_loss = evaluate_on_validation(encoder, decoder, val_loader,
criterion, device)
print(f"Epoch [{epoch+1}/{num_epochs}] finished. Avg Train Loss:
{avg_loss:.4f} | Val Loss: {avg_val_loss:.4f}")

if avg_val_loss < best_val_loss:
    best_val_loss = avg_val_loss
    epochs_no_improve = 0
    torch.save(encoder.state_dict(), 'encoder_best.pth')
    torch.save(decoder.state_dict(), 'decoder_best.pth')
    print("Validation loss improved, model saved!")
else:
    epochs_no_improve += 1
    print(f"No improvement in validation loss for {epochs_no_improve}
epoch(s).")
    if epochs_no_improve >= patience:
        print("Early stopping triggered.")
        break
```

- Evaluates model on validation set after each epoch.
- Saves best model if validation loss improves.
- Implements early stopping if no improvement for patience epochs.
- Purpose: Prevents overfitting and saves best mode

## 14. Checkpoint Saving

```
print("Training complete!")
```

- Prints completion message after training loop.



## 15. Checkpoint Saving

```
torch.save({
    'epoch': epoch,
    'encoder_state_dict': encoder.state_dict(),
    'decoder_state_dict': decoder.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'best_val_loss': best_val_loss,
    'epochs_no_improve': epochs_no_improve
}, 'checkpoint.pth')
```

- Saves model, optimizer, and training state after each epoch.
- Purpose: Allows resuming training and model recovery

## Techniques and Modern Processes

- Transfer Learning: Uses pretrained CNN (ResNet) for image feature extraction.
- Teacher Forcing: Uses ground truth captions as input during training for stable learning.
- Sequence Padding: Handles variable-length captions with padding.
- Early Stopping: Prevents overfitting by monitoring validation loss.
- Checkpointing: Saves model state for recovery and resumption.
- Device-Agnostic Code: Automatically uses GPU if available.
- Data Augmentation: Image transformations for robustness

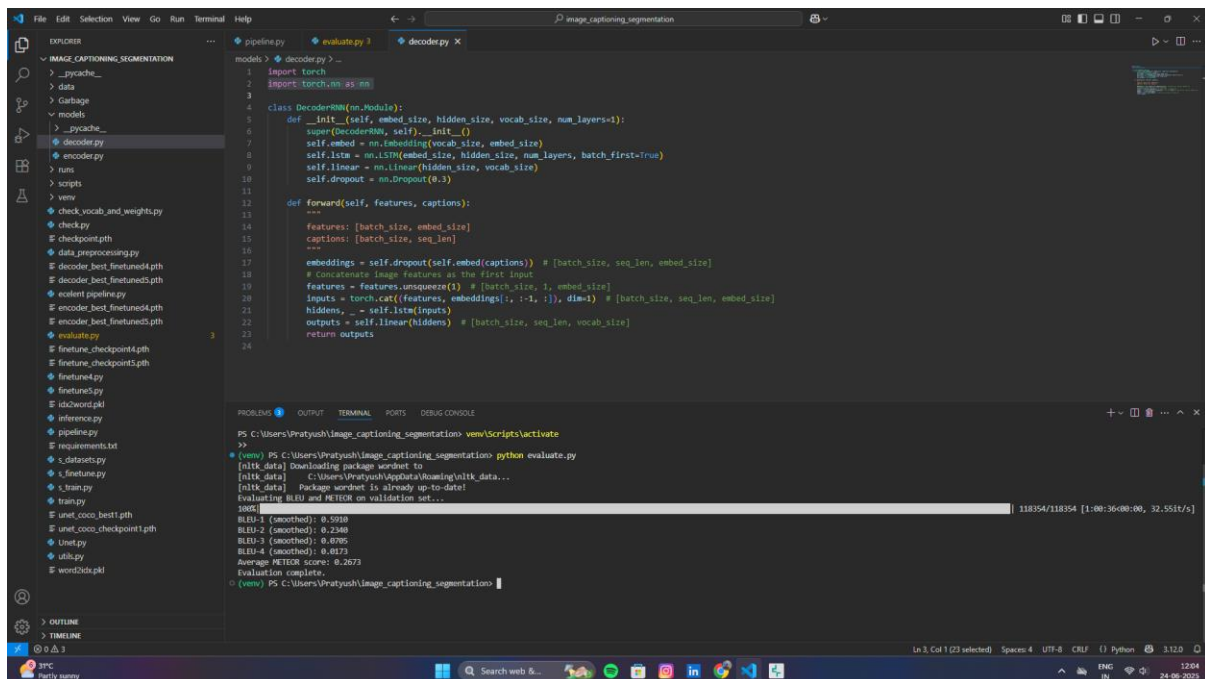
## • EVALUATION WITH BLEU AND METEOR MATRICES

This script is used to evaluate the performance of the model based on BLEU 1 - 4 scores and METEOR scores. It gives a clear idea of how the model will perform in real-time.

### KEY FEATURES :

- Data Handling & Pre-processing .
- Model Loading .
- Vocabulary & Token Handling .
- Caption Generation .
  - Implements **Beam Search Decoding** :
    - Beam size configurable (beam\_size=3).
    - Searches top probable sequences at each decoding step.
    - Uses decoder embedding + LSTM + linear layer.
- **Evaluation Metrics**
  - **BLEU Scores:**
    - BLEU-1 to BLEU-4 computed using nltk.translate.bleu\_score.
    - Applies **smoothing function** (method4) for stability.
  - **METEOR Score:**
    - Per-sample METEOR computed using nltk.translate.meteor\_score.
    - Averages across the dataset.
- **Performance Reporting**
  - Prints smoothed BLEU-1 to BLEU-4 scores.
  - Prints average METEOR score.
  - Uses tqdm to show progress bar over validation set.

# OUTPUT OF EVALUATION



```
model.py
1 import torch
2 import torch.nn as nn
3
4 class DecoderRNN(nn.Module):
5     def __init__(self, embed_size, hidden_size, vocab_size, num_layers=1):
6         super(DecoderRNN, self).__init__()
7         self.embed = nn.Embedding(vocab_size, embed_size)
8         self.lstm = nn.LSTM(embed_size, hidden_size, num_layers, batch_first=True)
9         self.linear = nn.Linear(hidden_size, vocab_size)
10        self.dropout = nn.Dropout(0.3)
11
12    def forward(self, features, captions):
13        """
14        features: [batch_size, embed_size]
15        captions: [batch_size, seq_len]
16        """
17        embeddings = self.dropout(self.embed(captions)) # [batch_size, seq_len, embed_size]
18        # Concatenate image features as the first input
19        features = features.unsqueeze(1) # [batch_size, 1, embed_size]
20        inputs = torch.cat((features, embeddings), dim=1) # [batch_size, seq_len, embed_size]
21        hidden, _ = self.lstm(inputs)
22        outputs = self.linear(hidden) # [batch_size, seq_len, vocab_size]
23        return outputs
24
```

```
PS C:\Users\Pratyush\image_captioning_segmentation> python evaluate.py
>
[nltk_data] Downloading package wordnet to
[nltk_data] C:\Users\Pratyush\AppData\Local\nltk_data...
[nltk_data] Package wordnet is already up-to-date!
Evaluating BLEU and METEOR on validation set...
118354/118354 [1:00:36:00:00, 32.55it/s]
BLEU-1 (smoothed): 0.5910
BLEU-2 (smoothed): 0.2340
BLEU-3 (smoothed): 0.0705
BLEU-4 (smoothed): 0.0173
Average METEOR score: 0.2673
Evaluation complete.
PS C:\Users\Pratyush\image_captioning_segmentation>
```

## Evaluation Results :

From the terminal output in the screenshot, my model produced the following metrics:

### 🕒 BLEU Scores (Smoothed)

- **BLEU-1:** 0.5910
- **BLEU-2:** 0.2340
- **BLEU-3:** 0.0705
- **BLEU-4:** 0.0173

### 🕒 METEOR Score

- **Average METEOR:** 0.2673

**BLEU-1** is very high (0.59) → The model is good at predicting individual words correctly (usually common nouns, verbs). **BLEU-2 to BLEU-4** drop sharply → Indicates poor performance on **longer n-gram sequences**, meaning:

- Poor grammar or fluency.
- Weakness in predicting the correct sequence or structure of sentences.

**METEOR** Score is 0.2673 moderate . **METEOR** accounts for synonymy and word order better than BLEU. This suggests that even though the phrasing may differ from references, the model captures meaning to some extent.

**Inference** : Due to non-attention mechanism and non-transformer based decoder we can say this is an average model .

- **FINETUNING THE MODEL WITH LOWERED LEARNING RATES AND ADJUSTED DECODER DROPOUT**

In this Phase I adjusted the learning rates to various values and checked if any part I can improve model performance or not and the decoder dropout was also increased to 0.3 - 0.5 .

The learning rates were adjusted and scheduler was also added.

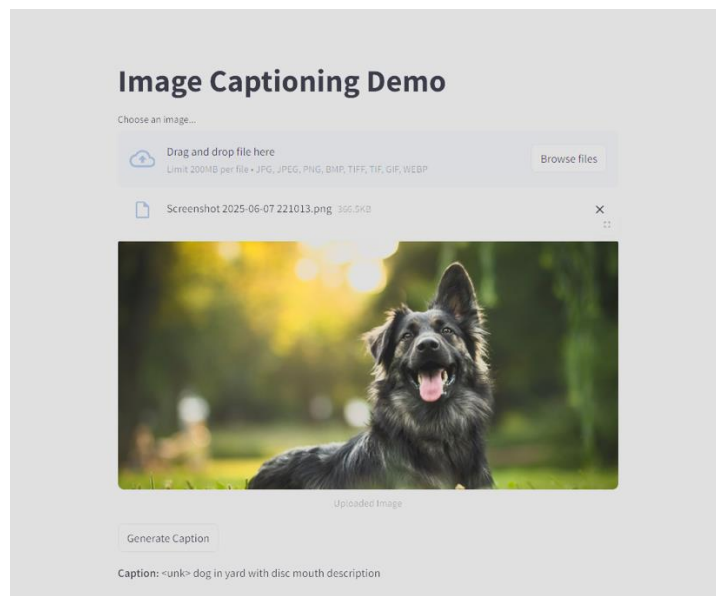
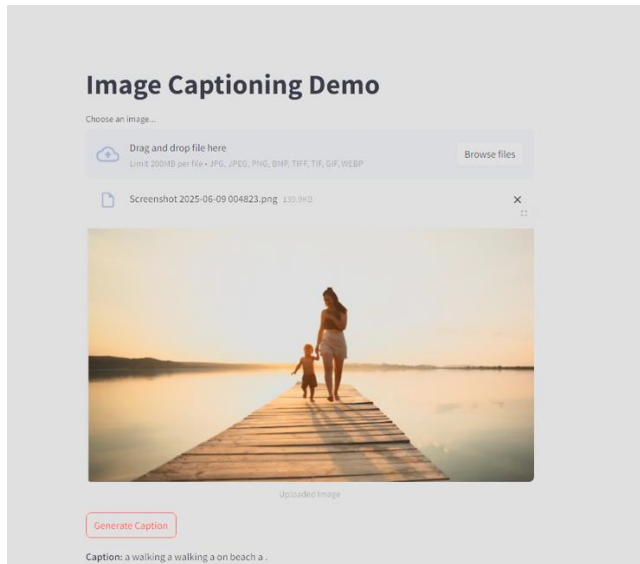
Rest of the training script remained same and finally my Early stopping triggered to prevent over-fitting of my model.

```
Epoch [21/30] Loss: 2.7168: 100%  
Epoch [21/30] finished. Avg Train Loss: 3.0083 | Val Loss: 2.9990  
No improvement in validation loss for 8 epoch(s).  
Early stopping triggered.  
Training complete!
```

Final Validation loss - 2.9990

Final Train Loss - 3.0083

- **DEPLOYING ON STREAMLIT LOCAL HOST FOR VISUALIZATION**



THESE ARE ALL BEFORE FINETUNING AND BEFORE ENDING THE TRAINING SCRIPT.

## II. PROCEEDING WITH IMAGE SEGMENTATION

### • DATA PRE-PROCESSING:

This custom PyTorch Dataset class:

- Loads images and their corresponding segmentation masks from the COCO dataset.
- Prepares them for training segmentation models like U-Net, Mask R-CNN, etc.

Core Components:

`__init__()` – Initialization

- `root`: Path to COCO images folder.
- `annFile`: Path to COCO annotation .json file.
- `transforms`: Optional transforms to apply to images.
- `image_size`: Target size for images and masks (default: 128×128).
- `n_classes`: Number of classes (default: 81 including background).

Category Mapping:

- Creates a mapping: `category_id` → `train_id` (for segmentation mask labels), where:
  - 0 = background
  - 1...N = class IDs for objects

`__getitem__()` – Data Retrieval

For each image:

1. Load Image:
  - Retrieves image and opens it with PIL.
  - Resizes to `image_size`.
2. Load Annotations:
  - Retrieves all annotations for that image (object masks).
  - For each object:
    - Converts segmentation annotation to binary mask.
    - Resizes the mask.
    - Combines all object masks into one final mask (per-pixel class labels).
3. Apply Transforms:
  - Converts image to tensor (with optional additional transforms).

- Converts mask to a LongTensor.

4. Returns:

python

CopyEdit

(image\_tensor, segmentation\_mask\_tensor)

`__len__()` – Dataset Length

- Returns the total number of images in the dataset.

### **Output Example:**

Each call to `dataset[i]` returns:

- Image: [3, H, W] (RGB image tensor)
- Mask: [H, W] with class labels per pixel (0 = background, others = object classes)

## • **MODELLING THE ARCHITECTURE OF U-NET**

### **Model Type:**

U-Net with ResNet34 backbone

- Designed for semantic segmentation tasks.
- Encoder is based on pretrained ResNet34 (`torchvision.models.resnet34`).
- Decoder is a typical U-Net-style upsampling path with skip connections.

Component Breakdown :

### **Encoder (Downsampling Path)**

- Extracts features at different resolutions using ResNet34 layers:
  - encoder1: Conv + BatchNorm + ReLU
  - encoder2: MaxPool + first residual block
  - encoder3-5: Remaining residual blocks
- ResNet34 provides deep and high-quality feature representations.

### Center Block (Bottleneck)

- Two convolutional layers with ReLU after the deepest encoder output (e5).
- Acts like U-Net's "bridge" before upsampling.

### Decoder (Upsampling Path)

- Series of up\_block() layers that:
  - Use ConvTranspose2d to upsample the feature maps.
  - Followed by standard conv+ReLU to refine.
- Each decoder layer is added to the corresponding encoder feature map (skip connection).

### Final Output Layer

- A 1x1 convolution to reduce channels to n\_classes.
- Upsample resizes the final output to match the input image/mask size (128x128).

### Forward Pass Flow

1. Image passes through each encoder stage: e1 → e5
2. Goes through bottleneck: center
3. Upsampling with skip connections: d5 → d2
4. Final segmentation logits: out = self.final(d2)
5. Output resized to 128×128 using bilinear interpolation.

### Use Case:

This model is suited for:

- Semantic segmentation on datasets like COCO, PASCAL VOC, medical images, etc.
- Especially useful when paired with datasets like my CocoSegmentationDataset.



- **CREATING THE TRAINING SCRIPT**

Train a semantic segmentation model (UnetResNet) on **COCO 2017** dataset using PyTorch, with support for:

- **Checkpointing**
- **Early stopping**
- **Mean IoU evaluation**

### **Key Components**

- **Setup**
  - Uses **GPU** if available.
  - Defines constants: `n_classes=81`, `image_size=(128, 128)`, `num_epochs=100`, etc.
  - Paths for saving:
    - `checkpoint_path`: for periodic saving
    - `best_model_path`: for best-performing model based on validation IoU
- **Data Loading**
  - Uses your custom `CocoSegmentationDataset` for both **train** and **validation**.
  - Applies resizing and tensor conversion via `torchvision.transforms`.
  - Loads data using PyTorch `DataLoader` with `pin_memory` for performance.
- **Model Initialization**
  - Initializes the **UnetResNet** model.
  - Loss: `nn.CrossEntropyLoss()` (standard for segmentation tasks).
  - Optimizer: Adam with learning rate `1e-4`.

- **Resume from Checkpoint (if exists)**
    - If `checkpoint_path` exists:
      - Loads model, optimizer, epoch number, and best IoU.
      - Resumes training from that epoch.
- 

- **Training Loop**

For each epoch:

- **Training phase:**
  - Forward pass → Compute loss → Backward pass → Optimizer step.
  - Tracks loss for reporting.
- **Validation phase:**
  - Model set to `eval()`.
  - Computes **mean IoU** using a `mean_iou()` utility function.
  - Prints training loss and validation IoU.

- **Checkpointing & Early Stopping**

- Saves:
  - A **checkpoint** after every epoch.
  - The **best model** (highest mean IoU).
- Implements **early stopping**:
  - If no improvement in IoU for 10 consecutive epochs → training stops early.

- **Final Output**

- At the end, prints the **best IoU achieved** during training.

```
def main():
    images, masks = images_loader(images_loader_path, masks_loader_path, num_classes=num_classes)
    outputs = model(images)
    iou = mean_iou(outputs, masks, num_classes)
    iou_scores.append(iou)
    val_bar.set_postfix(iou=iou)
    mean_val_iou = np.mean(iou_scores)
    print(f"Epoch {epoch+1}/{num_epochs} | Train Loss: {train_loss:.4f} | Mean IoU: {mean_val_iou:.4f}")

    # Save checkpoint
    checkpoint = {
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'best_iou': best_iou,
        'epochs_no_improve': epochs_no_improve,
    }
    torch.save(checkpoint, checkpoint_path)

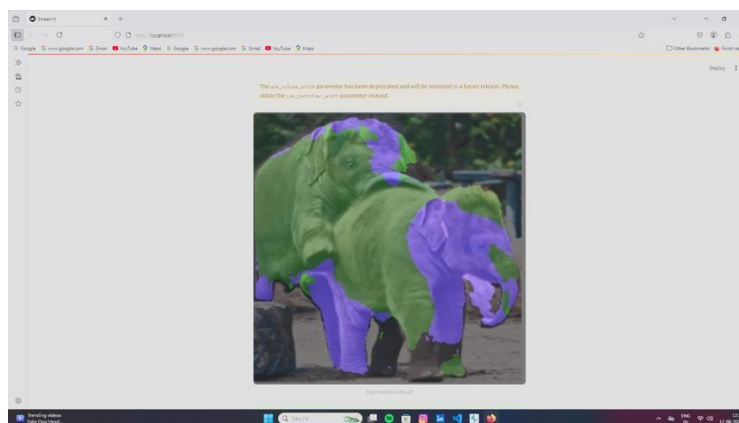
    # Save best model and update early stopping
    if mean_val_iou > best_iou:
        best_iou = mean_val_iou
        torch.save(model.state_dict(), best_model_path)
        print(f"Best model saved at epoch {epoch+1} with IoU: {best_iou:.4f}")
        epochs_no_improve = 0
    else:
        epochs_no_improve += 1
        print(f"No improvement for {epochs_no_improve} epoch(s).")

    # Early stopping
    if epochs_no_improve == 10:
        print("Early stopping: No improvement in IoU for 10 consecutive epochs. Training complete. Best IoU achieved: 0.1922173782720085.")
        break
```

Using device: cuda  
GPU: NVIDIA GeForce RTX 4080 Ti  
Loading annotations into memory...  
Done (t=0.77s)  
creating index...  
Index created!  
Loading annotations into memory...  
Done (t=0.38s)  
creating index...  
Index created!  
Resuming from checkpoint: unet\_coco\_checkpoint1.pth  
Resumed from epoch 17, best IoU so far: 0.1922  
Epoch 38/100 | Train Loss: 0.2215 | Mean IoU: 0.1921  
No improvement for 10 epoch(s).  
Early stopping: No improvement in IoU for 10 consecutive epochs.  
Training complete. Best IoU achieved: 0.1922173782720085.  
(venv) PS C:\Users\Pratyush\image\_captioning\_segmentation

Early stopping triggered with the best IoU value of 0.19

## • DEPLOYING ON STREAMLIT FOR VISUALIZATION



Visualizing the results in between training with s\_app.py , This is not the final output just a normal visualization.

- **PIPELINING:**

- **Integrating Captioning Model. (Trained on default Image-net Weights).**

```
# --- Load Caption Generation Model ---
try:
    # NOTE: Ensure 'inference.py' and its dependencies are in the project directory.
    from inference import generate_caption
except ImportError:
    st.warning("⚠️ Caption generation module not found. Please ensure 'inference.py' is in the project directory. Captioning will be disabled.")
def generate_caption(image_path):
    # Simple caption generation as fallback
    captions = [
        "A beautiful scene captured with intricate details and vibrant colors.",
        "An image showcasing various objects and elements in a natural setting.",
        "A photograph displaying rich textures and interesting visual composition.",
        "A scene with multiple elements creating a harmonious visual narrative.",
        "An image featuring diverse objects and subjects in their environment.",
        "A captivating photograph with excellent lighting and composition.",
        "A detailed image showing various subjects in their natural context.",
    ]
    return random.choice(captions)
```

- **Mask R-CNN and Faster R-CNN for Object and Instance detection (Pretrained Models).**

```
# --- Load Mask R-CNN Model ---
@st.cache_resource
def load_maskrcnn():
    """Loads the Mask R-CNN model for instance segmentation."""
    try:
        # Load pre-trained Mask R-CNN model
        model = torchvision.models.detection.maskrcnn_resnet50_fpn(weights='DEFAULT')
        model.eval()
        model.to(DEVICE)
        st.success("✅ Mask R-CNN model loaded successfully!")
        return model
    except Exception as e:
        st.error(f"❌ Error loading Mask R-CNN model: {e}")
        return None

# --- Load Object Detection Model ---
@st.cache_resource
def load_object_detection_model():
    """Loads the object detection model (Faster R-CNN)."""
    try:
        # Load pre-trained Faster R-CNN model
        model = torchvision.models.detection.fasterrcnn_resnet50_fpn(weights='DEFAULT')
        model.eval()
        model.to(DEVICE)
        st.success("✅ Object detection model loaded successfully!")
        return model
    except Exception as e:
        st.error(f"❌ Error loading object detection model: {e}")
        return None
```

- **U-net model. (Trained on default Image-net Weights)**

```
# --- Load U-Net Model ---
@st.cache_resource
def load_unet():
    """Loads the U-Net model for semantic segmentation."""
    try:
        from Unet import UNetResNet
        model = UNetResNet(n_classes=N_CLASSES, out_size=IMAGE_SIZE)
        # NOTE: Ensure 'unet_coco_best1.pth' is in your project's root directory.
        model.load_state_dict(torch.load("unet_coco_best1.pth", map_location=DEVICE))
        model.eval()
        model.to(DEVICE)
        return model
    except ImportError:
        st.warning("⚠️ U-Net model definition not found. Please ensure 'Unet.py' is in the project directory. Segmentation feature will be disabled.")
        return None
    except FileNotFoundError:
        st.warning("⚠️ U-Net weights file ('unet_coco_best1.pth') not found. Segmentation feature will be disabled.")
        return None
    except Exception as e:
        st.error(f"❌ Error loading U-Net model: {e}")
        return None
```

# USE CASES

## 1. Accessibility for Visually Impaired

- Describe images in real-time using generated captions.
- Assistive tech like Seeing AI and Be My Eyes use similar models.

## 2. Smart Photo Management

- Auto-tagging photos in personal or cloud libraries (Google Photos, iCloud).
- Enables search like “photos with a dog playing with a ball”.

## 3. E-Commerce Automation

- Automatically generate product descriptions for listings.
- Helps when uploading thousands of new images daily.

## 4. Journalism & Media Archiving

- Captioning large image datasets from events or news footage.
- Saves manual tagging and improves searchability.

## 5. Content Moderation & Surveillance

- Describe and classify content in sensitive areas (e.g., public safety).
- Captions help flag inappropriate or unusual scenes.

## 6. Robotics and Autonomous Systems

- Robots interpreting environments and narrating actions (“red ball on the table”).

## Use Cases of Image Segmentation Model

Semantic segmentation classifies every pixel into categories (e.g., person, road, car). Your U-Net + ResNet model on COCO can power:

## 1. Autonomous Driving (ADAS)

- Segment roads, pedestrians, vehicles, and lane markings.
- Essential for environment perception in self-driving cars.

## 2. Medical Imaging

- Segment tumors, organs, or lesions in X-rays, CT scans, MRI.
- Used in cancer detection, surgical planning, and diagnosis.

## 3. Satellite and Aerial Image Analysis

- Land use classification: segment buildings, vegetation, water, etc.
- Urban planning, deforestation monitoring, disaster analysis.

## 4. Virtual Try-On / AR Shopping

- Precisely segment clothes, faces, or spectacles to try products in AR.
- Used in apps like Lenskart, IKEA Place.

## 5. Robotic Vision / Object Manipulation

- Segment objects for picking, sorting, or tracking in factory robotics.

## 6. Agriculture

- Segment crops, weeds, or pests from drone or soil images.
- Enables precision farming and early disease detection.

### Combined Use Case (Captioning + Segmentation)

When combined, both models can be powerful in:

Smart Visual Assistants

“A man riding a bike on the street” (caption)

- segmented objects → used for contextual reasoning.

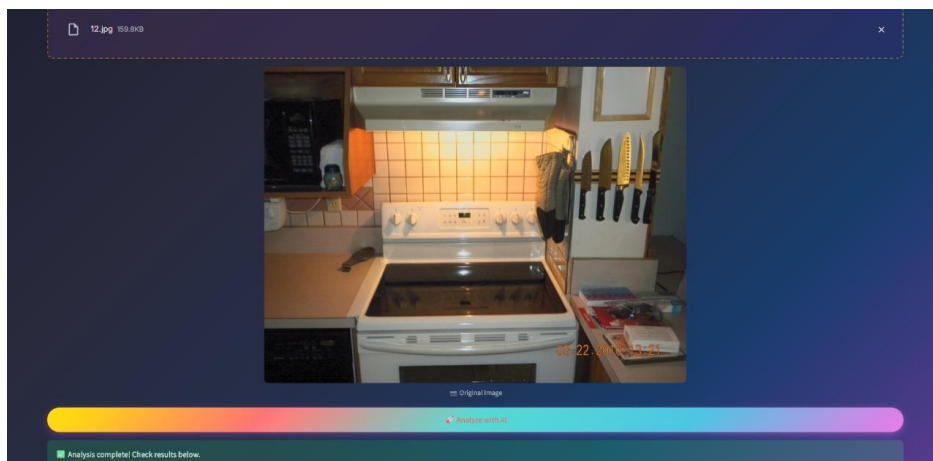
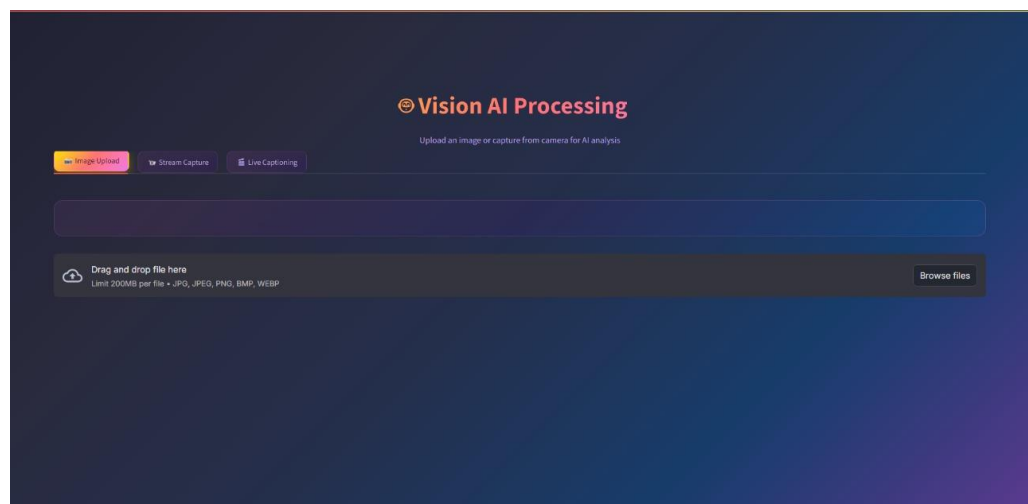
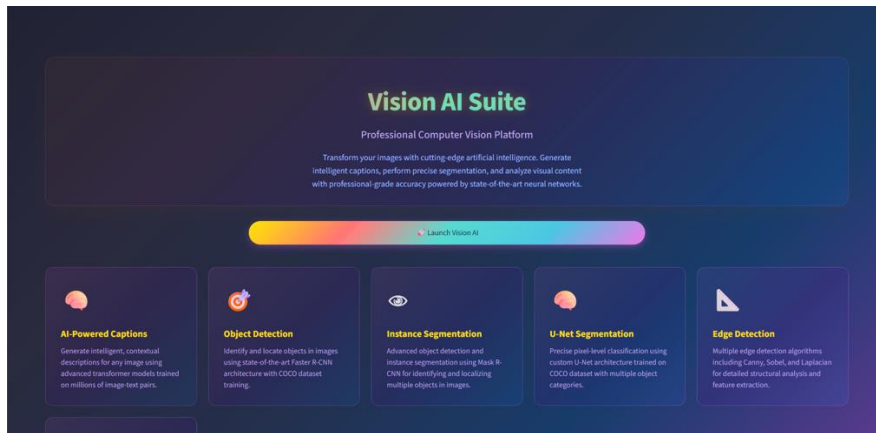
Multi-Modal Search Engines

- Query: “Show images where a person is holding a phone near a dog”.
- Captioning + segmentation enables powerful visual-textual search.

# FUTURE IMPROVEMENTS

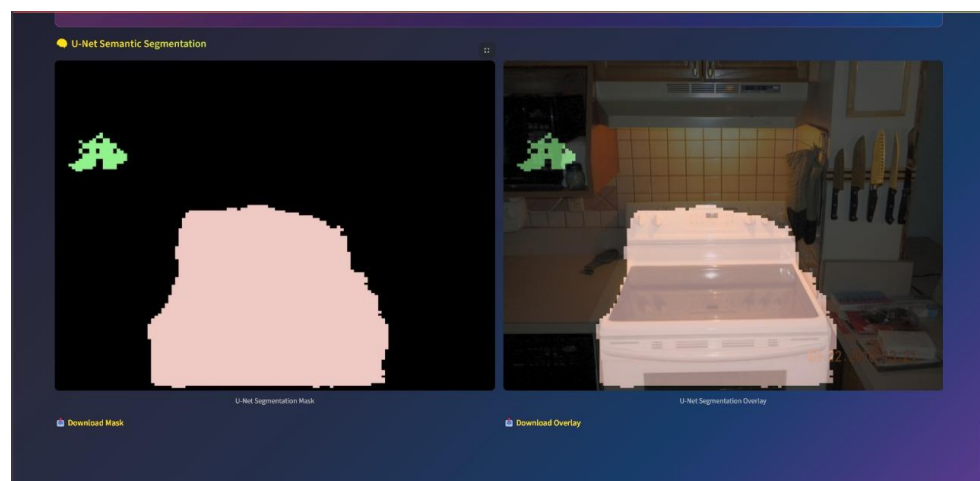
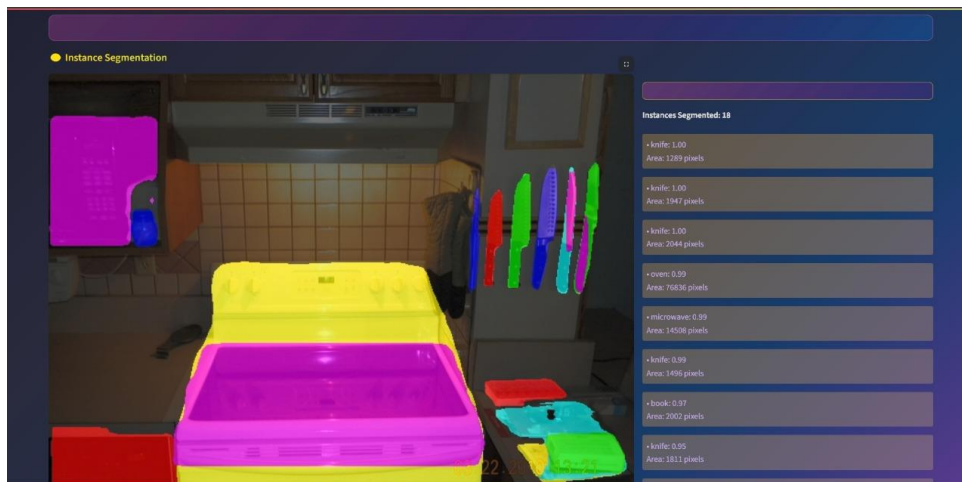
- Modifying the encoder and decoder architectures, with transformer architectures.
- Implementing attention mechanism.
- Training for more number of epochs.

# FINAL OUTPUT









**LIVE CAMERA CAPTIONING IS ALSO THERE  
PLEASE FIND THE FULL DEMONSTRATION OF  
THE PROJECT :-**

**<https://drive.google.com/file/d/1nLHmGm1nOJevUNe9IPMIO8DzURGyKqhm/view?usp=drivesdk>**