Name - Piyush Kumar Gupta

Btech 3rd Year

```
!pip install --upgrade transformers
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.11/dist-packages (4.51.3)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from transformers) (3.18.0)
Requirement already satisfied: huggingface-hub<1.0,>=0.30.0 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.31.2)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.11/dist-packages (from transformers) (2.0.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from transformers) (24.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-packages (from transformers) (6.0.2)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.11/dist-packages (from transformers) (2024.11.6)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (from transformers) (2.32.3)
Requirement already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.21.1)
Requirement already satisfied: safetensors>=0.4.3 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.5.3)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.11/dist-packages (from transformers) (4.67.1)
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.11/dist-packages (from huggingface-hub<1.0,>=0.30.0->trans
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.11/dist-packages (from huggingface-hub<1.0,>=0.3
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests->transformers) (3
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests->transformers) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests->transformers) (2.4.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests->transformers) (2025.4.2
```

```
from torch.utils.data import Dataset, DataLoader
from torch.cuda import amp
import torch
from torch import nn
from sklearn.metrics import accuracy_score
from tqdm.notebook import tqdm
import os
from PIL import Image
from transformers import SegformerForSemanticSegmentation, SegformerFeatureExtractor
import pandas as pd
import cv2
import numpy as np
import albumentations as aug
```

```
/usr/local/lib/python3.11/dist-packages/albumentations/__init__.py:28: UserWarning: A new version of Albumentations is available: '2
  check_for_updates()
```

```
from albumentations import Compose, HorizontalFlip, VerticalFlip

transform = Compose([
    HorizontalFlip(p=0.5),  # For horizontal flips
    VerticalFlip(p=0.5)   # For vertical flips (uncomment if needed)
])
```

```
WIDTH = 256
HEIGHT = 256
```

```
# from torch.utils.data import Dataset
# import os
# import cv2
# import albumentations as aug

# class ImageSegmentationDataset(Dataset):
#     """Image segmentation dataset."""

#     def __init__(self, root_dir, feature_extractor, transforms=None, split="train"):
#         """
#         Args:
#             root_dir (string): Root directory of the dataset.
#             feature_extractor (SegFormerFeatureExtractor): Feature extractor.
#             transforms (albumentations.Compose): Data augmentations.
#             split (string): "train", "val", or "test" to indicate the split.
#         """
#         self.root_dir = root_dir
#         self.feature_extractor = feature_extractor
#         self.transforms = transforms
#         self.split = split

#         # Assuming images in 'images/train' and masks in 'mask/train'
#         self.img_dir = os.path.join(self.root_dir, "source")
```

```
#         self.ann_dir = os.path.join(self.root_dir, "masks")

#         # Read image and annotation file names
#         self.images = sorted(os.listdir(self.img_dir))
#         self.annotations = sorted(os.listdir(self.ann_dir))

#         assert len(self.images) == len(self.annotations), "Unequal number of images and masks"

#     def __len__(self):
#         return len(self.images)

#     def __getitem__(self, idx):

#         image_path = os.path.join(self.img_dir, self.images[idx])
#         mask_path = os.path.join(self.ann_dir, self.annotations[idx])

#         image = cv2.imread(image_path)
#         image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

#         segmentation_map = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
#         image = cv2.resize(image, (WIDTH, HEIGHT), interpolation=cv2.INTER_LINEAR)
#         segmentation_map = cv2.resize(segmentation_map, (WIDTH, HEIGHT), interpolation=cv2.INTER_NEAREST)

#         # Apply transforms based on split
#         if self.split == "train" and self.transforms is not None:
#             augmented = self.transforms(image=image, mask=segmentation_map)
#             image, segmentation_map = augmented['image'], augmented['mask']

#         encoded_inputs = self.feature_extractor(image, segmentation_map, return_tensors="pt")

#         # Remove batch dimension
#         for k, v in encoded_inputs.items():
#             encoded_inputs[k].squeeze_()

#         return encoded_inputs
```

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

```
Mounted at /content/drive
```

```
from torch.utils.data import Dataset
import os
import cv2
import albumentations as aug

class ImageSegmentationDataset(Dataset):
    """Image segmentation dataset."""

    def __init__(self, root_dir, feature_extractor, transforms=None, split="train", image_size=256):
        """
        Args:
            root_dir (string): Root directory of the dataset.
            feature_extractor (SegFormerFeatureExtractor): Feature extractor.
            transforms (albumentations.Compose): Data augmentations.
            split (string): "train", "val", or "test" to indicate the split.
            image_size (int): Desired size for resizing images (default: 256).
        """
        self.root_dir = root_dir
        self.feature_extractor = feature_extractor
        self.transforms = transforms
        self.split = split
        self.image_size = image_size  # Store image size

        # Assuming images in 'images/train' and masks in 'mask/train'
        self.img_dir = os.path.join(self.root_dir, "source")
        self.ann_dir = os.path.join(self.root_dir, "masks")

        # Read image and annotation file names
        self.images = sorted(os.listdir(self.img_dir))
        self.annotations = sorted(os.listdir(self.ann_dir))

        assert len(self.images) == len(self.annotations), "Unequal number of images and masks"

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):

        image_path = os.path.join(self.img_dir, self.images[idx])
        mask_path = os.path.join(self.ann_dir, self.annotations[idx])
```

```
        image = cv2.imread(image_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        segmentation_map = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)

        # Resize images to the specified image_size
        image = cv2.resize(image, (self.image_size, self.image_size), interpolation=cv2.INTER_LINEAR)
        segmentation_map = cv2.resize(segmentation_map, (self.image_size, self.image_size), interpolation=cv2.INTER_NEAREST)

        # Apply transforms based on split
        if self.split == "train" and self.transforms is not None:
            augmented = self.transforms(image=image, mask=segmentation_map)
            image, segmentation_map = augmented['image'], augmented['mask']

        encoded_inputs = self.feature_extractor(image, segmentation_map, return_tensors="pt")

        # Remove batch dimension
        for k, v in encoded_inputs.items():
            encoded_inputs[k].squeeze_()

        return encoded_inputs
```

```
from torch.utils.data import random_split
```

```
import torch
from torch.utils.data import random_split, DataLoader
from albumentations import Compose, HorizontalFlip, VerticalFlip
from transformers import SegformerFeatureExtractor

# Set seed for reproducibility
seed = 42
torch.manual_seed(seed)

# Data augmentation transforms
transform = Compose([
    HorizontalFlip(p=0.5),
    VerticalFlip(p=0.5)
])

# Define root directory and feature extractor
root_dir = '/content/drive/MyDrive/Colab Notebooks/Segmentation/filtered_dataset'
feature_extractor = SegformerFeatureExtractor(size=256, align=False, reduce_zero_label=False)

# Create the full dataset
dataset = ImageSegmentationDataset(root_dir=root_dir, feature_extractor=feature_extractor, transforms=transform)

# Split the dataset with a fixed seed
dataset_size = len(dataset)
train_size = int(0.7 * dataset_size)
val_size = int(0.15 * dataset_size)
test_size = dataset_size - train_size - val_size

generator = torch.Generator().manual_seed(seed)
train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size, val_size, test_size], generator=generator)

# Create data loaders
train_dataloader = DataLoader(train_dataset, batch_size=4, shuffle=True)
valid_dataloader = DataLoader(val_dataset, batch_size=4)
test_dataloader = DataLoader(test_dataset, batch_size=4)
```

```
/usr/local/lib/python3.11/dist-packages/transformers/models/segformer/feature_extraction_segformer.py:28: FutureWarning: The class S
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/transformers/utils/deprecation.py:172: UserWarning: The following named arguments are not va
  return func(*args, **kwargs)
```

```
print("Number of training examples:", len(train_dataset))
print("Number of validation examples:", len(val_dataset))
```

```
Number of training examples: 3323
Number of validation examples: 712
```

```
encoded_inputs = train_dataset[0]
```

```
encoded_inputs["pixel_values"].shape
```

```
torch.Size([3, 256, 256])
```

```
encoded_inputs["labels"].shape
```

```
torch.Size([256, 256])
```

```
encoded_inputs["labels"]
```

```
tensor([[0, 0, 0,  ..., 0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0],
        ...,
        [0, 0, 0,  ..., 0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0]])
```

```
encoded_inputs["labels"].squeeze().unique()
```

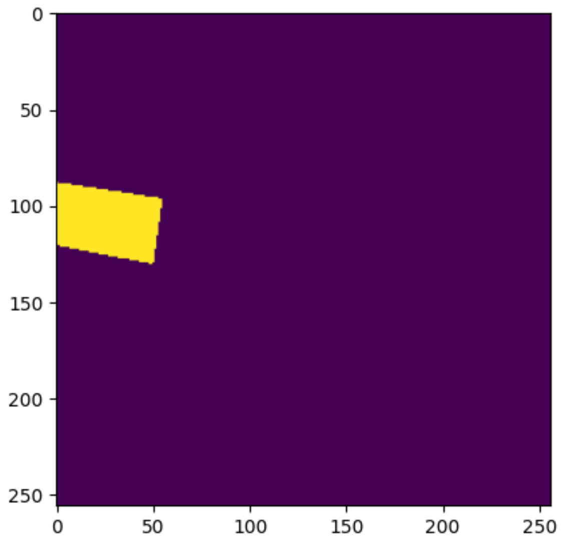
```
tensor([0, 1])
```

```
mask = encoded_inputs["labels"].numpy()
```

```
import matplotlib.pyplot as plt
```

```
plt.imshow(mask)
```

```
<matplotlib.image.AxesImage at 0x7cbb376e3cd0>
```




```
batch = next(iter(train_dataloader))
```

```
for k,v in batch.items():
    print(k, v.shape)
```

```
pixel_values torch.Size([4, 3, 256, 256])
labels torch.Size([4, 256, 256])
```

```
batch["labels"].shape
```

```
torch.Size([4, 256, 256])
```

```
# Assuming your dataset has 2 classes (0 for background, 1 for building)
id2label = {0: "background", 1: "building"}
label2id = {v: k for k, v in id2label.items()}

# Update model initialization
model = SegformerForSemanticSegmentation.from_pretrained(
    "nvidia/mit-b0",
    ignore_mismatched_sizes=True,
    num_labels=len(id2label),  # Set num_labels to the number of classes (2 in this case)
    id2label=id2label,
    label2id=label2id,
    reshape_last_stage=True,
    image_size=256
)
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as :
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
```

config.json: 100% 70.0k/70.0k [00:00<00:00, 2.96MB/s]

pytorch_model.bin: 100% 14.4M/14.4M [00:00<00:00, 87.1MB/s]

```
Some weights of SegformerForSemanticSegmentation were not initialized from the model checkpoint at nvidia/mit-b0 and are newly init:
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

```
!pip install transformers
from torch.optim import AdamW # Import AdamW from transformers
```

Show hidden output

```
optimizer = AdamW(model.parameters(), lr=0.00006)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print("Model Initialized!")
```

```
Model Initialized!
```

```
#!pip install torch>=2.0
```

Training Model- no Need To execute Everytime

```
criterion = nn.CrossEntropyLoss()  # Example using CrossEntropyLoss
accumulation_steps = 8
scaler = amp.GradScaler()

for epoch in range(1, 11):  # loop over the dataset multiple times
    print("Epoch:", epoch)
    pbar = tqdm(train_dataloader)
    accuracies = []
    losses = []
    val_accuracies = []
    val_losses = []
    model.train()
    for idx, batch in enumerate(pbar):
        # get the inputs;
        pixel_values = batch["pixel_values"].to(device)
        labels = batch["labels"].to(device)

        print("Before modification:")
        print("Minimum label value:", labels.min().item())
        print("Maximum label value:", labels.max().item())
        # Ensure labels only contain 0 and 1 (binary segmentation)
        labels = labels.long()  # Cast labels to Long type
        labels[labels > 1] = 0  # Force any values > 1 to be 0 (background)

        # zero the parameter gradients
        optimizer.zero_grad()

        # ---start of changes---
        # forward pass
        outputs = model(pixel_values=pixel_values, labels=labels)

        # interpolate the logits to the same size as the labels
        upsampled_logits = nn.functional.interpolate(
            outputs.logits, size=labels.shape[-2:], mode="bilinear", align_corners=False
        )
        # ---end of changes---

        with torch.cuda.amp.autocast():
            # calculate loss with upsampled logits
            loss = criterion(upsampled_logits, labels)
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        # evaluate (use upsampled logits here as well)
        predicted = upsampled_logits.argmax(dim=1)
```

```
        # Calculate loss using the weighted criterion
        loss = criterion(upsampled_logits, labels)  # Use upsampled logits here as well
        loss = loss / accumulation_steps

        mask = (labels != 255) # we don't include the background class in the accuracy calculation
        pred_labels = predicted[mask].detach().cpu().numpy()
        true_labels = labels[mask].detach().cpu().numpy()
        accuracy = accuracy_score(pred_labels, true_labels)
        accuracies.append(accuracy)
        losses.append(loss.item())
        pbar.set_postfix({'Batch': idx, 'Pixel-wise accuracy': sum(accuracies)/len(accuracies), 'Loss': sum(losses)/len(losses)})

        # backward + optimize
        if (idx + 1) % accumulation_steps == 0:
            optimizer.step()                            # Now we can do an optimizer step
            optimizer.zero_grad()

    else:
      model.eval()
      with torch.no_grad():
        for idx, batch in enumerate(valid_dataloader):
          pixel_values = batch["pixel_values"].to(device)
          labels = batch["labels"].to(device)

          # Ensure labels only contain 0 and 1 for validation as well
          labels = labels.long()
          labels[labels > 1] = 0

          outputs = model(pixel_values=pixel_values, labels=labels)
          upsampled_logits = nn.functional.interpolate(outputs.logits, size=labels.shape[-2:], mode="bilinear", align_corners=False)
          predicted = upsampled_logits.argmax(dim=1)
          mask = (labels != 255) # we don't include the background class in the accuracy calculation
          pred_labels = predicted[mask].detach().cpu().numpy()
          true_labels = labels[mask].detach().cpu().numpy()
          accuracy = accuracy_score(pred_labels, true_labels)
          val_loss = outputs.loss
          val_accuracies.append(accuracy)
          val_losses.append(val_loss.item())
    print(f"Train Pixel-wise accuracy: {sum(accuracies)/len(accuracies)}\
         Train Loss: {sum(losses)/len(losses)}\
         Val Pixel-wise accuracy: {sum(val_accuracies)/len(val_accuracies)}\
         Val Loss: {sum(val_losses)/len(val_losses)}")
```

Show hidden output

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print("Model Initialized!")
```

```
Model Initialized!
```

Training part No need to execute every time

```
import torch
from torch.cuda.amp import GradScaler, autocast
import torch.optim as optim

# Use CPU as the device
device = torch.device('cpu')

# Initialize GradScaler, but without using AMP since we're on CPU
scaler = GradScaler(enabled=False)  # No AMP on CPU

# Define the optimizer (SGD in your case)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

for epoch in range(1, 11):  # loop over the dataset multiple times
    print("Epoch:", epoch)
    pbar = tqdm(train_dataloader)
    accuracies = []
    losses = []
    val_accuracies = []
    val_losses = []
    val_ious = []  # To store IoU values for validation
    model.train()

    for idx, batch in enumerate(pbar):
        # get the inputs
        pixel_values = batch["pixel_values"].to(device)
```

```
            labels = batch["labels"].to(device)

            # Ensure labels only contain 0 and 1 (binary segmentation)
            labels = labels.long()  # Cast labels to Long type
            labels[labels > 1] = 0  # Force any values > 1 to be 0 (background)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward pass
            outputs = model(pixel_values=pixel_values, labels=labels)

            # interpolate the logits to the same size as the labels
            upsampled_logits = nn.functional.interpolate(
                outputs.logits, size=labels.shape[-2:], mode="bilinear", align_corners=False
            )

            # No AMP: Direct loss calculation
            loss = criterion(upsampled_logits, labels)

            loss.backward()
            optimizer.step()

            # evaluate (use upsampled logits here as well)
            predicted = upsampled_logits.argmax(dim=1)

            # Calculate loss using the weighted criterion
            loss = criterion(upsampled_logits, labels)  # Use upsampled logits here as well
            loss = loss / accumulation_steps

            mask = (labels != 255)  # we don't include the background class in the accuracy calculation
            pred_labels = predicted[mask].detach().cpu().numpy()
            true_labels = labels[mask].detach().cpu().numpy()
            accuracy = accuracy_score(pred_labels, true_labels)
            accuracies.append(accuracy)
            losses.append(loss.item())

            # Calculate IoU for this batch
            iou_value = compute_iou(predicted, labels)
            val_ious.append(iou_value.item())  # Store IoU for this batch

            pbar.set_postfix({'Batch': idx, 'Pixel-wise accuracy': sum(accuracies)/len(accuracies), 'Loss': sum(losses)/len(losses)})

        # Validation phase
        else:
            model.eval()
            with torch.no_grad():
                for idx, batch in enumerate(valid_dataloader):
                    pixel_values = batch["pixel_values"].to(device)
                    labels = batch["labels"].to(device)

                    # Ensure labels only contain 0 and 1 for validation as well
                    labels = labels.long()
                    labels[labels > 1] = 0

                    outputs = model(pixel_values=pixel_values, labels=labels)
                    upsampled_logits = nn.functional.interpolate(outputs.logits, size=labels.shape[-2:], mode="bilinear", align_corners=Fal
                    predicted = upsampled_logits.argmax(dim=1)

                    mask = (labels != 255)  # we don't include the background class in the accuracy calculation
                    pred_labels = predicted[mask].detach().cpu().numpy()
                    true_labels = labels[mask].detach().cpu().numpy()
                    accuracy = accuracy_score(pred_labels, true_labels)
                    val_loss = outputs.loss
                    val_accuracies.append(accuracy)
                    val_losses.append(val_loss.item())

                    # Calculate IoU for the validation batch
                    iou_value = compute_iou(predicted, labels)
                    val_ious.append(iou_value.item())

        print(f"Train Pixel-wise accuracy: {sum(accuracies)/len(accuracies)}\
             Train Loss: {sum(losses)/len(losses)}\
             Train IoU: {sum(val_ious)/len(val_ious)}\
             Val Pixel-wise accuracy: {sum(val_accuracies)/len(val_accuracies)}\
             Val Loss: {sum(val_losses)/len(val_losses)}\
             Val IoU: {sum(val_ious)/len(val_ious)}")
```

Show hidden output

```
import torch

def compute_iou(pred, target, num_classes=2):
    """
    Compute the Intersection over Union (IoU) for a binary or multi-class segmentation task.

    Args:
        pred (Tensor): The predicted tensor of shape [batch_size, height, width]
        target (Tensor): The ground truth tensor of shape [batch_size, height, width]
        num_classes (int): Number of classes in the segmentation task (default is 2 for binary)

    Returns:
        iou (Tensor): The IoU for each class, of shape [num_classes]
    """
    iou = torch.zeros(num_classes).to(pred.device)

    for cls in range(num_classes):
        # Create binary masks for each class (foreground = 1, background = 0)
        pred_class = (pred == cls).float()
        target_class = (target == cls).float()

        # Compute intersection and union for this class
        intersection = (pred_class * target_class).sum()
        union = pred_class.sum() + target_class.sum() - intersection

        # Avoid division by zero by ensuring the union is not zero
        iou[cls] = intersection / (union + 1e-6)  # Adding epsilon to prevent divide-by-zero errors

    return iou.mean()  # Return the mean IoU over all classes
```

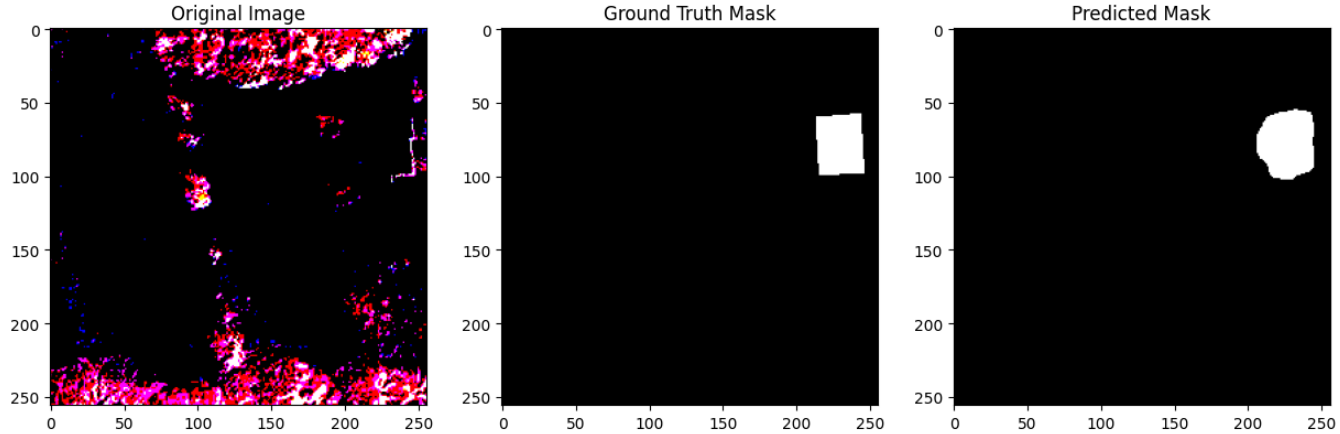
```
# prompt: open results of model stored at /content/drive/MyDrive/saved_models/segformer_model.pth

model_path = "/content/drive/MyDrive/Colab Notebooks/Segmentation/segformer_model.pth"
model.load_state_dict(torch.load(model_path))
print(f"Model loaded from {model_path}")
```

```
Model loaded from /content/drive/MyDrive/Colab Notebooks/Segmentation/segformer_model.pth
```

```
# prompt: now show results of this saved model, take any one random imgae  , show me accuracy and loss of predicted output too

import torch
import random
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np

# Assuming you have the necessary variables and functions defined from the previous code
# ... (including model, feature_extractor, device, etc.)

# Choose a random image from the test dataset
random_image_index = random.randint(0, len(test_dataset) - 1)
encoded_inputs = test_dataset[random_image_index]

# Move inputs to the device
pixel_values = encoded_inputs["pixel_values"].unsqueeze(0).to(device)  # Add batch dimension
labels = encoded_inputs["labels"].unsqueeze(0).to(device)


# Ensure labels only contain 0 and 1 for validation as well
labels = labels.long()
labels[labels > 1] = 0

# Perform inference
with torch.no_grad():
    model.eval()
    outputs = model(pixel_values=pixel_values, labels=labels)
    upsampled_logits = nn.functional.interpolate(
        outputs.logits, size=labels.shape[-2:], mode="bilinear", align_corners=False
    )
    predicted = upsampled_logits.argmax(dim=1)

# Calculate accuracy and loss
mask = (labels != 255)
pred_labels = predicted[mask].detach().cpu().numpy()
true_labels = labels[mask].detach().cpu().numpy()
accuracy = accuracy_score(pred_labels, true_labels)
loss = outputs.loss.item()
```

```
print(f"Accuracy: {accuracy}")
print(f"Loss: {loss}")

# Display the original image, ground truth mask, and predicted mask
original_image = Image.fromarray(np.transpose(pixel_values.squeeze(0).cpu().numpy(), (1, 2, 0)).astype(np.uint8))
ground_truth_mask = Image.fromarray(labels.squeeze(0).cpu().numpy().astype(np.uint8))
predicted_mask = Image.fromarray(predicted.squeeze(0).cpu().numpy().astype(np.uint8))

plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.title("Original Image")
plt.imshow(original_image)

plt.subplot(1, 3, 2)
plt.title("Ground Truth Mask")
plt.imshow(ground_truth_mask, cmap="gray")

plt.subplot(1, 3, 3)
plt.title("Predicted Mask")
plt.imshow(predicted_mask, cmap="gray")

plt.show()
```

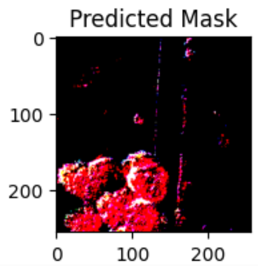
```
Accuracy: 0.993927001953125
Loss: 0.016645636409521103
```



Try to Coorect

```
# Choose a random image from the test dataset
random_image_index = random.randint(0, len(test_dataset) - 1)
encoded_inputs = test_dataset[random_image_index]

plt.subplot(1, 3, 3)
plt.title("Predicted Mask")

plt.imshow(encoded_inputs['pixel_values'].squeeze(0).cpu().numpy().transpose(1, 2, 0).astype('uint8'))
```

```
<matplotlib.image.AxesImage at 0x7cbb2e191090>
```




```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
# prompt: give code to test the stored model on test data , and store result of each epoch in a excel file, keep remember to include iou

import pandas as pd
from sklearn.metrics import jaccard_score

# ... (your existing code) ...

# Create an empty list to store results
results = []

# Test loop
model.eval()
with torch.no_grad():
    for idx, batch in enumerate(test_dataloader):
        pixel_values = batch["pixel_values"].to(device)
        labels = batch["labels"].to(device)

        # Ensure labels only contain 0 and 1 for testing as well
        labels = labels.long()
        labels[labels > 1] = 0

        outputs = model(pixel_values=pixel_values, labels=labels)
        upsampled_logits = nn.functional.interpolate(
            outputs.logits, size=labels.shape[-2:], mode="bilinear", align_corners=False
        )
        predicted = upsampled_logits.argmax(dim=1)

        # Calculate accuracy
        mask = (labels != 255)
        pred_labels = predicted[mask].detach().cpu().numpy()
        true_labels = labels[mask].detach().cpu().numpy()
        accuracy = accuracy_score(pred_labels, true_labels)

        # Calculate IoU
        iou = jaccard_score(true_labels, pred_labels, average='macro') # or 'weighted'

        loss = outputs.loss.item()

        results.append({
            'Image Index': idx,
            'Accuracy': accuracy,
            'IoU': iou,
            'Loss': loss
        })

# Create a pandas DataFrame from the results
results_df = pd.DataFrame(results)

# Save results to an Excel file
results_df.to_excel('test_results.xlsx', index=False)
print("Test results saved to test_results.xlsx")
```

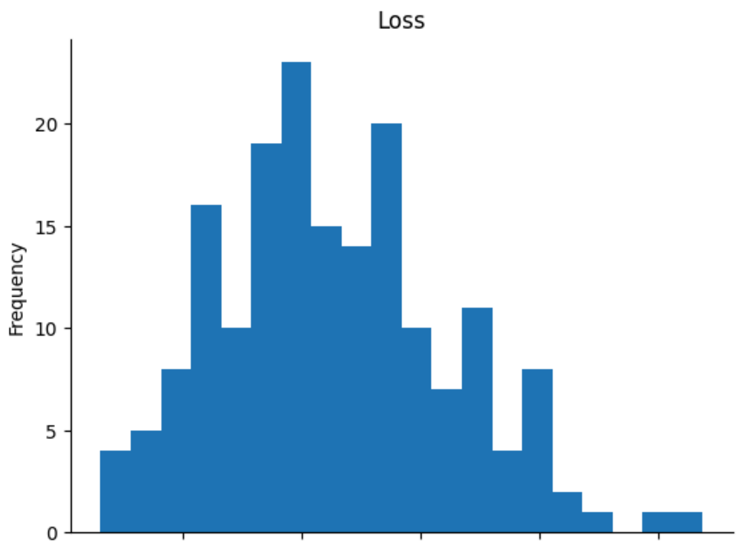
Show hidden output

```
# prompt: show test results

# Assuming you have the necessary variables and functions defined from the previous code
# ... (including model, feature_extractor, device, etc.)

# Create an empty list to store results
results = []

# Test loop
model.eval()
with torch.no_grad():
    for idx, batch in enumerate(test_dataloader):
        pixel_values = batch["pixel_values"].to(device)
        labels = batch["labels"].to(device)

        # Ensure labels only contain 0 and 1 for testing as well
        labels = labels.long()
        labels[labels > 1] = 0

        outputs = model(pixel_values=pixel_values, labels=labels)
        upsampled_logits = nn.functional.interpolate(
            outputs.logits, size=labels.shape[-2:], mode="bilinear", align_corners=False
        )
        predicted = upsampled_logits.argmax(dim=1)

        # Calculate accuracy
        mask = (labels != 255)
```

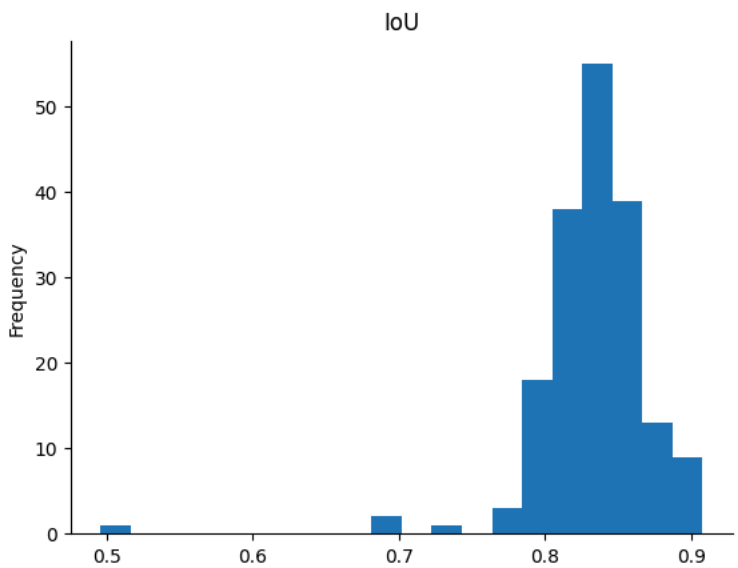
```
        pred_labels = predicted[mask].detach().cpu().numpy()
        true_labels = labels[mask].detach().cpu().numpy()
        accuracy = accuracy_score(pred_labels, true_labels)

        # Calculate IoU
        iou = jaccard_score(true_labels, pred_labels, average='macro') # or 'weighted'

        loss = outputs.loss.item()

        results.append({
            'Image Index': idx,
            'Accuracy': accuracy,
            'IoU': iou,
            'Loss': loss
        })

# Create a pandas DataFrame from the results
results_df = pd.DataFrame(results)

# Display the results DataFrame
results_df
```

Show hidden output

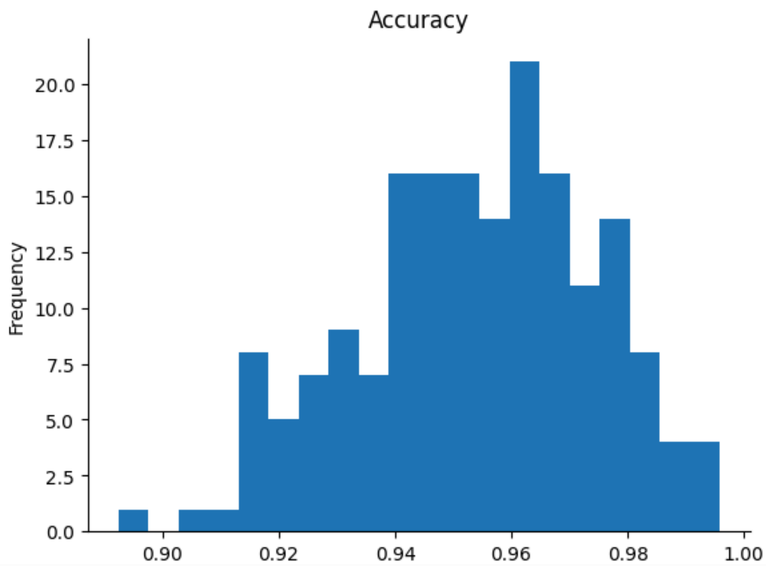
```
from matplotlib import pyplot as plt
results_df['Loss'].plot(kind='hist', bins=20, title='Loss')
plt.gca().spines[['top', 'right',]].set_visible(False)
```



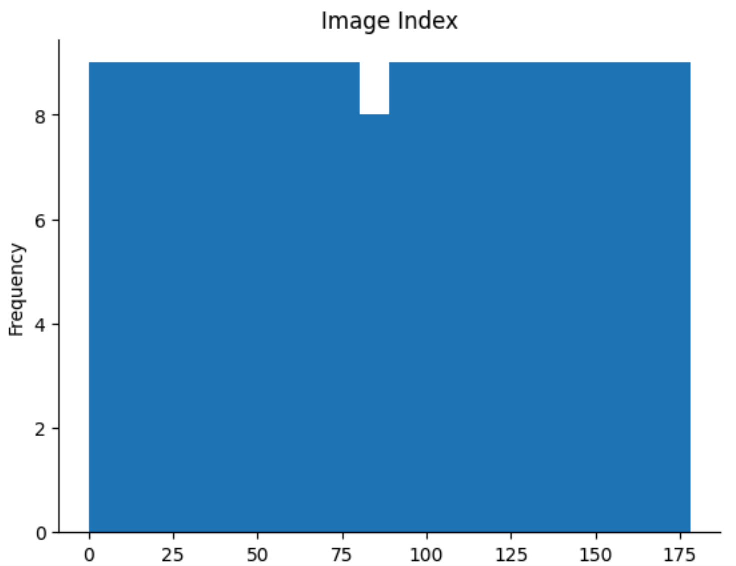

```
from matplotlib import pyplot as plt
results_df['IoU'].plot(kind='hist', bins=20, title='IoU')
plt.gca().spines[['top', 'right',]].set_visible(False)
```



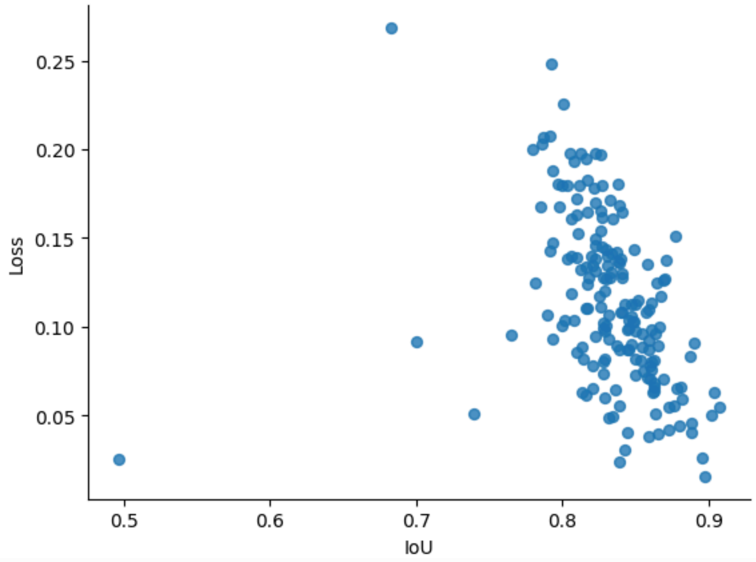
```
from matplotlib import pyplot as plt
results_df['Accuracy'].plot(kind='hist', bins=20, title='Accuracy')
plt.gca().spines[['top', 'right',]].set_visible(False)
```



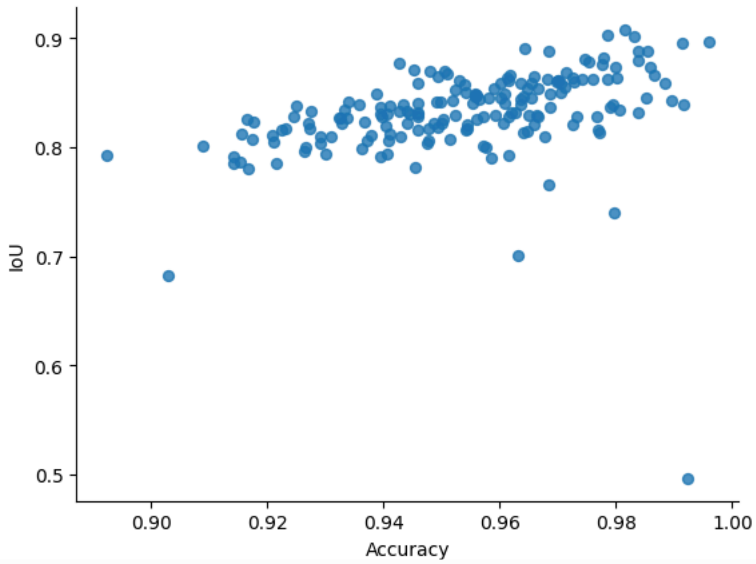

```
from matplotlib import pyplot as plt
results_df['Image Index'].plot(kind='hist', bins=20, title='Image Index')
plt.gca().spines[['top', 'right',]].set_visible(False)
```



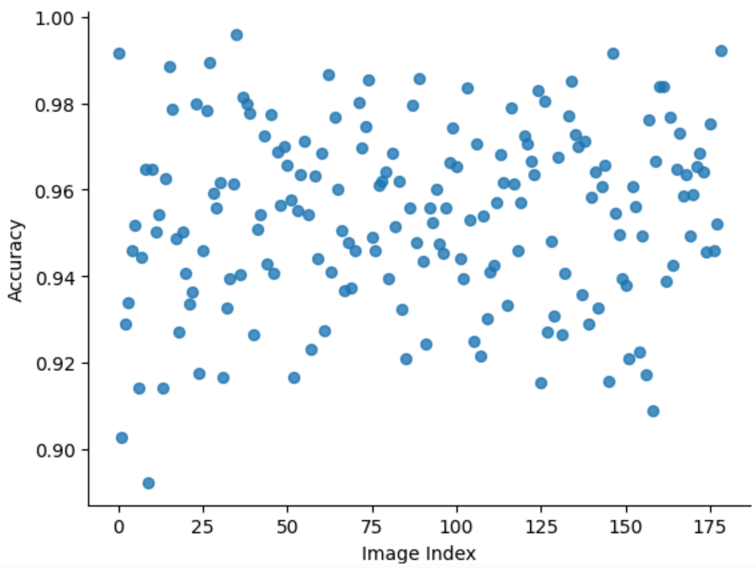

```
from matplotlib import pyplot as plt
results_df.plot(kind='scatter', x='IoU', y='Loss', s=32, alpha=.8)
plt.gca().spines[['top', 'right',]].set_visible(False)
```



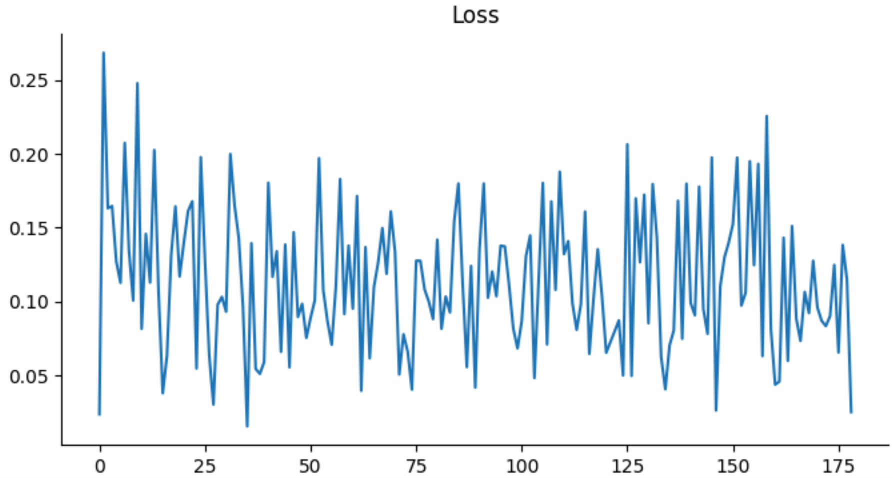

```
from matplotlib import pyplot as plt
results_df.plot(kind='scatter', x='Accuracy', y='IoU', s=32, alpha=.8)
plt.gca().spines[['top', 'right',]].set_visible(False)
```



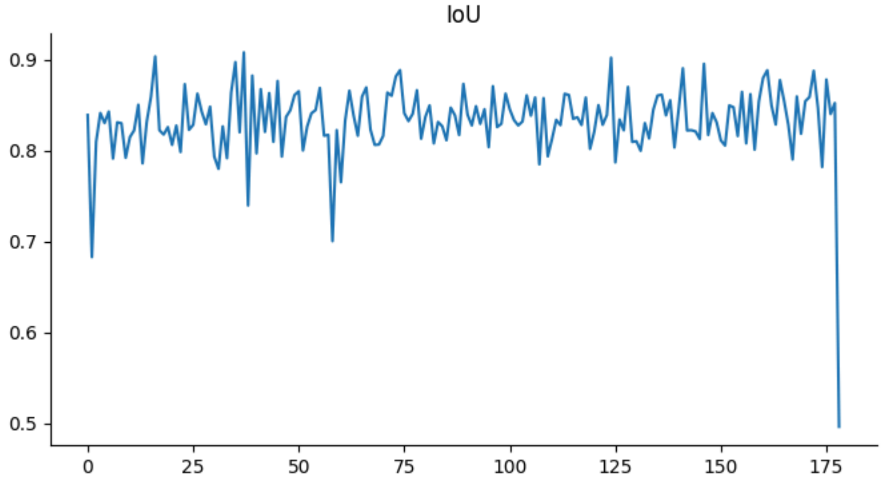

```
from matplotlib import pyplot as plt
results_df.plot(kind='scatter', x='Image Index', y='Accuracy', s=32, alpha=.8)
plt.gca().spines[['top', 'right',]].set_visible(False)
```



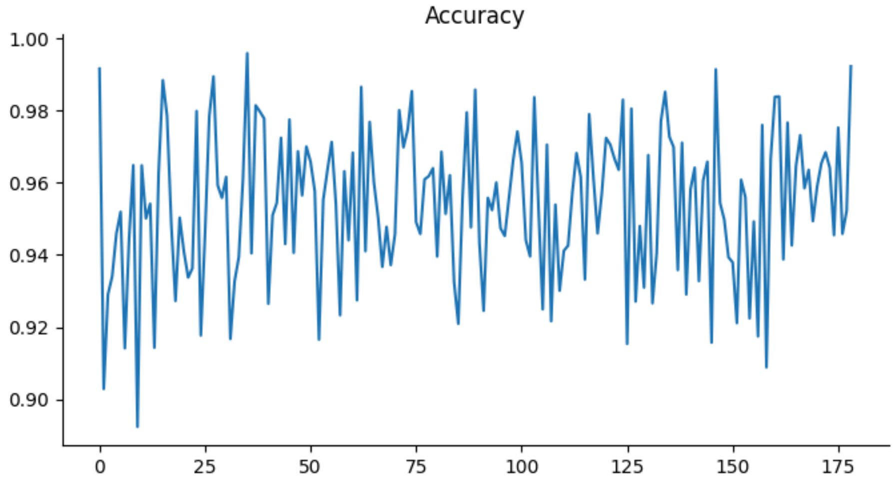
```
from matplotlib import pyplot as plt
results_df['Loss'].plot(kind='line', figsize=(8, 4), title='Loss')
plt.gca().spines[['top', 'right']].set_visible(False)
```



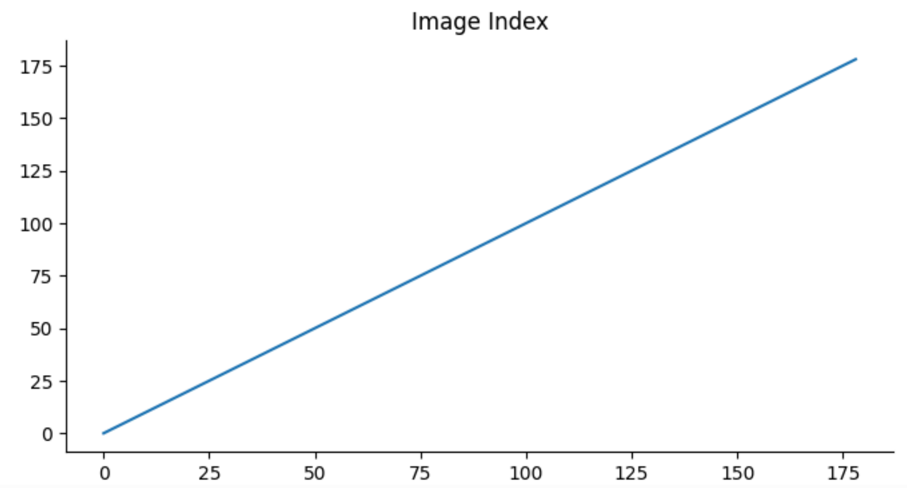

```
from matplotlib import pyplot as plt
results_df['IoU'].plot(kind='line', figsize=(8, 4), title='IoU')
plt.gca().spines[['top', 'right']].set_visible(False)
```



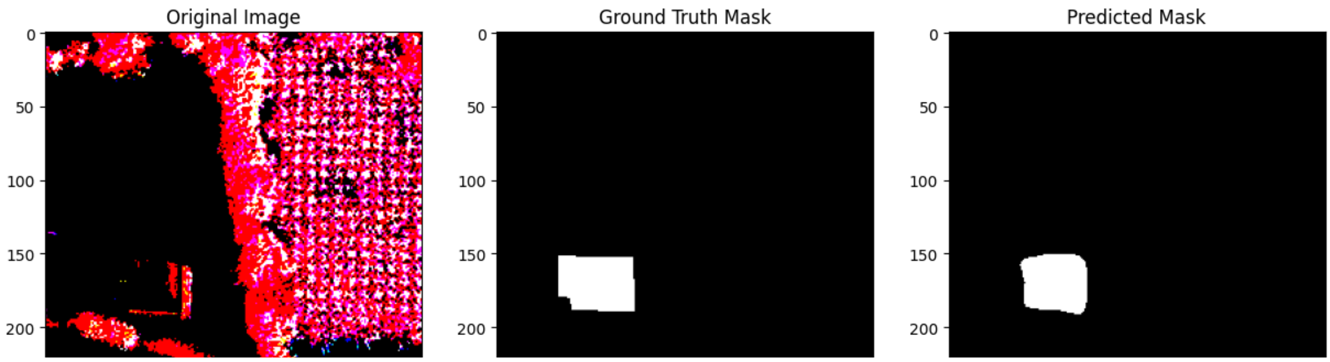

```
from matplotlib import pyplot as plt
results_df['Accuracy'].plot(kind='line', figsize=(8, 4), title='Accuracy')
plt.gca().spines[['top', 'right']].set_visible(False)
```




```
from matplotlib import pyplot as plt
results_df['Image Index'].plot(kind='line', figsize=(8, 4), title='Image Index')
plt.gca().spines[['top', 'right']].set_visible(False)
```




```
# prompt: generate a graph on test data applied on my model

import matplotlib.pyplot as plt
import random

# ... (your existing code) ...

# Choose a random image from the test dataset
random_image_index = random.randint(0, len(test_dataset) - 1)
encoded_inputs = test_dataset[random_image_index]

# Move inputs to the device
pixel_values = encoded_inputs["pixel_values"].unsqueeze(0).to(device)  # Add batch dimension
labels = encoded_inputs["labels"].unsqueeze(0).to(device)

# Ensure labels only contain 0 and 1 for testing as well
labels = labels.long()
labels[labels > 1] = 0

# Perform inference
with torch.no_grad():
    model.eval()
    outputs = model(pixel_values=pixel_values, labels=labels)
    upsampled_logits = nn.functional.interpolate(
        outputs.logits, size=labels.shape[-2:], mode="bilinear", align_corners=False
    )
    predicted = upsampled_logits.argmax(dim=1)

# ... (rest of your existing code for calculations and saving results) ...


# Display the original image, ground truth mask, and predicted mask
original_image = Image.fromarray(np.transpose(pixel_values.squeeze(0).cpu().numpy(), (1, 2, 0)).astype(np.uint8))
ground_truth_mask = Image.fromarray(labels.squeeze(0).cpu().numpy().astype(np.uint8))
predicted_mask = Image.fromarray(predicted.squeeze(0).cpu().numpy().astype(np.uint8))

plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.title("Original Image")
plt.imshow(original_image)

plt.subplot(1, 3, 2)
plt.title("Ground Truth Mask")
plt.imshow(ground_truth_mask, cmap="gray")

plt.subplot(1, 3, 3)
plt.title("Predicted Mask")
plt.imshow(predicted_mask, cmap="gray")

plt.show()
```




```
# prompt: generate a graph on test data applied on my model

import matplotlib.pyplot as plt
import random

# ... (your existing code) ...

# Choose a random image from the test dataset
random_image_index = random.randint(0, len(test_dataset) - 1)
encoded_inputs = test_dataset[random_image_index]

# Move inputs to the device
pixel_values = encoded_inputs["pixel_values"].unsqueeze(0).to(device)  # Add batch dimension
labels = encoded_inputs["labels"].unsqueeze(0).to(device)

# Ensure labels only contain 0 and 1 for testing as well
# ***This is where the distortion might happen if your original masks have more than 2 classes***
# labels = labels.long()
# labels[labels > 1] = 0

# Perform inference
with torch.no_grad():
    model.eval()
    outputs = model(pixel_values=pixel_values, labels=labels)
    upsampled_logits = nn.functional.interpolate(
        outputs.logits, size=labels.shape[-2:], mode="bilinear", align_corners=False
    )
    predicted = upsampled_logits.argmax(dim=1)

# ... (rest of your existing code for calculations and saving results) ...


# Display the original image, ground truth mask, and predicted mask
original_image = Image.fromarray(np.transpose(pixel_values.squeeze(0).cpu().numpy(), (1, 2, 0)).astype(np.uint8))
# ***Modified to keep original mask values***
ground_truth_mask = Image.fromarray(encoded_inputs["labels"].cpu().numpy().astype(np.uint8))
predicted_mask = Image.fromarray(predicted.squeeze(0).cpu().numpy().astype(np.uint8))

plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.title("Original Image")
plt.imshow(original_image)

plt.subplot(1, 3, 2)
plt.title("Ground Truth Mask")
plt.imshow(ground_truth_mask, cmap="gray") # You can adjust the colormap if needed

plt.subplot(1, 3, 3)
plt.title("Predicted Mask")
plt.imshow(predicted_mask, cmap="gray")

plt.show()
```