**Python Code for Ret-Seg Net Framework**

1. **Xception70 –backbone network**

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import transforms

from torch.utils.data import DataLoader, Dataset

from sklearn.model\_selection import KFold

# Define RetSegNet architecture

class RetSegNet(nn.Module):

def \_\_init\_\_(self):

super(RetSegNet, self).\_\_init\_\_()

# Initialize Encoder and Decoder

self.encoder = Encoder()

self.decoder = Decoder()

def forward(self, x):

# Forward pass through Encoder and Decoder

encoder\_output = self.encoder(x)

final\_output = self.decoder(encoder\_output)

return final\_output

# Define Encoder architecture

class Encoder(nn.Module):

def \_\_init\_\_(self):

super(Encoder, self).\_\_init\_\_()

# Initialize Xception70 and ASPP

self.xception70 = Xception70()

self.aspp = ASPP()

def forward(self, x):

# Forward pass through Xception70 and ASPP

xception\_output = self.xception70(x)

aspp\_output = self.aspp(xception\_output)

return aspp\_output

# Define Decoder architecture

class Decoder(nn.Module):

def \_\_init\_\_(self):

super(Decoder, self).\_\_init\_\_()

# Initialize 1x1 Convolution, 3x3 Convolution, and Upsampling

self.conv1x1 = nn.Conv2d(in\_channels=2048, out\_channels=256, kernel\_size=1)

self.conv3x3 = nn.Conv2d(in\_channels=256, out\_channels=128, kernel\_size=3, padding=1)

self.upsample = nn.Upsample(scale\_factor=2, mode='bilinear', align\_corners=True)

def forward(self, encoder\_output):

# Forward pass through 1x1 Convolution, 3x3 Convolution, and Upsampling

conv1x1\_output = self.conv1x1(encoder\_output)

concatenated\_features = torch.cat((conv1x1\_output, encoder\_output), dim=1)

conv3x3\_output = self.conv3x3(concatenated\_features)

final\_output = self.upsample(conv3x3\_output)

return final\_output

# Define Xception70 architecture

class Xception70(nn.Module):

def \_\_init\_\_(self):

super(Xception70, self).\_\_init\_\_()

# Define Xception70 layers

# (You need to define the architecture of Xception70 based on your requirements)

def forward(self, x):

# Forward pass through Xception70 layers

return x # Placeholder

# Define ASPP architecture

class ASPP(nn.Module):

def \_\_init\_\_(self):

super(ASPP, self).\_\_init\_\_()

# Define ASPP layers

# (You need to define the architecture of ASPP based on your requirements)

def forward(self, x):

# Forward pass through ASPP layers

return x # Placeholder

# Define dataset and dataloader (Assuming you have your own dataset class)

class CustomDataset(Dataset):

def \_\_init\_\_(self, transform=None):

# Initialize dataset

pass

def \_\_len\_\_(self):

# Return the total number of samples

pass

def \_\_getitem\_\_(self, idx):

# Get data sample

pass

# Define evaluation metrics functions

def intersection\_over\_union(pred, target):

intersection = (pred & target).float().sum((1, 2)) # Calculate intersection

union = (pred | target).float().sum((1, 2)) + 1e-6 # Calculate union

iou = (intersection / union) \* 100 # Calculate IoU (%)

return iou

def sensitivity(pred, target):

true\_positive = (pred & target).float().sum((1, 2)) # Calculate True Positives

actual\_positive = target.float().sum((1, 2)) + 1e-6 # Calculate Actual Positives

sensitivity = (true\_positive / actual\_positive) \* 100 # Calculate Sensitivity (%)

return sensitivity

# Define hyperparameters

batch\_size = 32

learning\_rate = 0.0002

image\_size = (564, 375)

num\_epochs = 10

num\_folds = 10

# Initialize dataset and dataloader

dataset = CustomDataset(transform=transforms.ToTensor())

dataloader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True)

# Define loss function and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(RetSegNet.parameters(), lr=learning\_rate)

# Implement k-fold cross-validation

kf = KFold(n\_splits=num\_folds)

for train\_index, test\_index in kf.split(dataset):

train\_loader = DataLoader(dataset[train\_index], batch\_size=batch\_size, shuffle=True)

test\_loader = DataLoader(dataset[test\_index], batch\_size=batch\_size, shuffle=True)

# Initialize RetSegNet

RetSegNet = RetSegNet()

# Training loop

for epoch in range(num\_epochs):

for inputs, labels in train\_loader:

optimizer.zero\_grad()

outputs = RetSegNet(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

# Evaluation

with torch.no\_grad():

iou\_list = []

sensitivity\_list = []

for inputs, labels in test\_loader:

outputs = RetSegNet(inputs)

# Calculate evaluation metrics

predicted\_labels = torch.argmax(outputs, dim=1)

target\_masks = (labels == 1)

iou = intersection\_over\_union(predicted\_labels, target\_masks)

sensitivity\_val = sensitivity(predicted\_labels, target\_masks)

iou\_list.append(iou)

sensitivity\_list.append(sensitivity\_val)

# Compute average metrics

avg\_iou = torch.mean(torch.cat(iou\_list))

avg\_sensitivity = torch.mean(torch.cat(sensitivity\_list))

estimate\_deviation = 100 - avg\_iou.item()

std\_deviation = torch.std(torch.cat(iou\_list))

print(f"Epoch {epoch+1}: IoU: {avg\_iou.item()} Sensitivity: {avg\_sensitivity.item()} Estimate Deviation: {estimate\_deviation} Standard Deviation: {std\_deviation.item()}")

1. **ResNet101 –backbone network**

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import transforms

from torch.utils.data import DataLoader, Dataset

from sklearn.model\_selection import KFold

import torchvision.models as models

# Define RetSegNet architecture

class RetSegNet(nn.Module):

def \_\_init\_\_(self, backbone):

super(RetSegNet, self).\_\_init\_\_()

# Initialize Encoder and Decoder

self.encoder = Encoder(backbone)

self.decoder = Decoder()

def forward(self, x):

# Forward pass through Encoder and Decoder

encoder\_output = self.encoder(x)

final\_output = self.decoder(encoder\_output)

return final\_output

# Define Encoder architecture

class Encoder(nn.Module):

def \_\_init\_\_(self, backbone):

super(Encoder, self).\_\_init\_\_()

# Use ResNet101 as backbone

self.backbone = backbone

self.backbone.fc = nn.Identity() # Remove final fully connected layer

def forward(self, x):

# Forward pass through ResNet101 backbone

return self.backbone(x)

# Define Decoder architecture

class Decoder(nn.Module):

def \_\_init\_\_(self):

super(Decoder, self).\_\_init\_\_()

# Initialize 1x1 Convolution, 3x3 Convolution, and Upsampling

self.conv1x1 = nn.Conv2d(in\_channels=2048, out\_channels=256, kernel\_size=1)

self.conv3x3 = nn.Conv2d(in\_channels=256, out\_channels=128, kernel\_size=3, padding=1)

self.upsample = nn.Upsample(scale\_factor=2, mode='bilinear', align\_corners=True)

def forward(self, encoder\_output):

# Forward pass through 1x1 Convolution, 3x3 Convolution, and Upsampling

conv1x1\_output = self.conv1x1(encoder\_output)

concatenated\_features = torch.cat((conv1x1\_output, encoder\_output), dim=1)

conv3x3\_output = self.conv3x3(concatenated\_features)

final\_output = self.upsample(conv3x3\_output)

return final\_output

# Define dataset and dataloader (Assuming you have your own dataset class)

class CustomDataset(Dataset):

def \_\_init\_\_(self, transform=None):

# Initialize dataset

pass

def \_\_len\_\_(self):

# Return the total number of samples

pass

def \_\_getitem\_\_(self, idx):

# Get data sample

pass

# Define evaluation metrics functions

def intersection\_over\_union(pred, target):

intersection = (pred & target).float().sum((1, 2)) # Calculate intersection

union = (pred | target).float().sum((1, 2)) + 1e-6 # Calculate union

iou = (intersection / union) \* 100 # Calculate IoU (%)

return iou

def sensitivity(pred, target):

true\_positive = (pred & target).float().sum((1, 2)) # Calculate True Positives

actual\_positive = target.float().sum((1, 2)) + 1e-6 # Calculate Actual Positives

sensitivity = (true\_positive / actual\_positive) \* 100 # Calculate Sensitivity (%)

return sensitivity

# Define hyperparameters

batch\_size = 32

learning\_rate = 0.0002

image\_size = (564, 375)

num\_epochs = 10

num\_folds = 10

# Initialize ResNet101 as backbone

resnet101 = models.resnet101(pretrained=True)

# Initialize dataset and dataloader

dataset = CustomDataset(transform=transforms.ToTensor())

dataloader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True)

# Define loss function and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(RetSegNet.parameters(), lr=learning\_rate)

# Implement k-fold cross-validation

kf = KFold(n\_splits=num\_folds)

for train\_index, test\_index in kf.split(dataset):

train\_loader = DataLoader(dataset[train\_index], batch\_size=batch\_size, shuffle=True)

test\_loader = DataLoader(dataset[test\_index], batch\_size=batch\_size, shuffle=True)

# Initialize RetSegNet

RetSegNet = RetSegNet(resnet101)

# Training loop

for epoch in range(num\_epochs):

RetSegNet.train()

total\_loss = 0.0

correct = 0

total = 0

for inputs, labels in train\_loader:

optimizer.zero\_grad()

outputs = RetSegNet(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

\_, predicted = outputs.max(1)

total += labels.size(0)

correct += predicted.eq(labels).sum().item()

# Calculate training loss and accuracy

train\_loss = total\_loss / len(train\_loader)

train\_accuracy = 100 \* correct / total

# Evaluation

with torch.no\_grad():

RetSegNet.eval()

iou\_list = []

sensitivity\_list = []

for inputs, labels in test\_loader:

outputs = RetSegNet(inputs)

# Calculate evaluation metrics

predicted\_labels = torch.argmax(outputs, dim=1)

target\_masks = (labels == 1)

iou = intersection\_over\_union(predicted\_labels, target\_masks)

sensitivity\_val = sensitivity(predicted\_labels, target\_masks)

iou\_list.append(iou)

sensitivity\_list.append(sensitivity\_val)

# Compute average metrics

avg\_iou = torch.mean(torch.cat(iou\_list))

avg\_sensitivity = torch.mean(torch.cat(sensitivity\_list))

estimate\_deviation = 100 - avg\_iou.item()

std\_deviation = torch.std(torch.cat(iou\_list))

print(f"Epoch {epoch+1}: Train Loss: {train\_loss:.4f}, Train Accuracy: {train\_accuracy:.2f}%, IoU: {avg\_iou.item():.2f}, Sensitivity: {avg\_sensitivity.item():.2f}, Estimate Deviation: {estimate\_deviation:.2f}, Standard Deviation: {std\_deviation.item():.2f}")

1. **Inception V3 –backbone network**

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import transforms

from torch.utils.data import DataLoader, Dataset

from sklearn.model\_selection import KFold

import torchvision.models as models

# Define RetSegNet architecture

class RetSegNet(nn.Module):

def \_\_init\_\_(self, backbone):

super(RetSegNet, self).\_\_init\_\_()

# Initialize Encoder and Decoder

self.encoder = Encoder(backbone)

self.decoder = Decoder()

def forward(self, x):

# Forward pass through Encoder and Decoder

encoder\_output = self.encoder(x)

final\_output = self.decoder(encoder\_output)

return final\_output

# Define Encoder architecture

class Encoder(nn.Module):

def \_\_init\_\_(self, backbone):

super(Encoder, self).\_\_init\_\_()

# Use InceptionV3 as backbone

self.backbone = backbone

self.backbone.fc = nn.Identity() # Remove final fully connected layer

def forward(self, x):

# Forward pass through InceptionV3 backbone

return self.backbone(x)

# Define Decoder architecture

class Decoder(nn.Module):

def \_\_init\_\_(self):

super(Decoder, self).\_\_init\_\_()

# Initialize 1x1 Convolution, 3x3 Convolution, and Upsampling

self.conv1x1 = nn.Conv2d(in\_channels=2048, out\_channels=256, kernel\_size=1)

self.conv3x3 = nn.Conv2d(in\_channels=256, out\_channels=128, kernel\_size=3, padding=1)

self.upsample = nn.Upsample(scale\_factor=2, mode='bilinear', align\_corners=True)

def forward(self, encoder\_output):

# Forward pass through 1x1 Convolution, 3x3 Convolution, and Upsampling

conv1x1\_output = self.conv1x1(encoder\_output)

concatenated\_features = torch.cat((conv1x1\_output, encoder\_output), dim=1)

conv3x3\_output = self.conv3x3(concatenated\_features)

final\_output = self.upsample(conv3x3\_output)

return final\_output

# Define dataset and dataloader (Assuming you have your own dataset class)

class CustomDataset(Dataset):

def \_\_init\_\_(self, transform=None):

# Initialize dataset

pass

def \_\_len\_\_(self):

# Return the total number of samples

pass

def \_\_getitem\_\_(self, idx):

# Get data sample

pass

# Define evaluation metrics functions

def intersection\_over\_union(pred, target):

intersection = (pred & target).float().sum((1, 2)) # Calculate intersection

union = (pred | target).float().sum((1, 2)) + 1e-6 # Calculate union

iou = (intersection / union) \* 100 # Calculate IoU (%)

return iou

def sensitivity(pred, target):

true\_positive = (pred & target).float().sum((1, 2)) # Calculate True Positives

actual\_positive = target.float().sum((1, 2)) + 1e-6 # Calculate Actual Positives

sensitivity = (true\_positive / actual\_positive) \* 100 # Calculate Sensitivity (%)

return sensitivity

# Define hyperparameters

batch\_size = 32

learning\_rate = 0.0002

image\_size = (564, 375)

num\_epochs = 10

num\_folds = 10

# Load pre-trained InceptionV3

inceptionv3 = models.inception\_v3(pretrained=True)

# Initialize dataset and dataloader

dataset = CustomDataset(transform=transforms.ToTensor())

dataloader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True)

# Define loss function and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(RetSegNet.parameters(), lr=learning\_rate)

# Implement k-fold cross-validation

kf = KFold(n\_splits=num\_folds)

for train\_index, test\_index in kf.split(dataset):

train\_loader = DataLoader(dataset[train\_index], batch\_size=batch\_size, shuffle=True)

test\_loader = DataLoader(dataset[test\_index], batch\_size=batch\_size, shuffle=True)

# Initialize RetSegNet

RetSegNet = RetSegNet(inceptionv3)

# Training loop

for epoch in range(num\_epochs):

RetSegNet.train()

total\_loss = 0.0

correct = 0

total = 0

for inputs, labels in train\_loader:

optimizer.zero\_grad()

outputs = RetSegNet(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

\_, predicted = outputs.max(1)

total += labels.size(0)

correct += predicted.eq(labels).sum().item()

# Calculate training loss and accuracy

train\_loss = total\_loss / len(train\_loader)

train\_accuracy = 100 \* correct / total

# Evaluation

with torch.no\_grad():

RetSegNet.eval()

iou\_list = []

sensitivity\_list = []

for inputs, labels in test\_loader:

outputs = RetSegNet(inputs)

# Calculate evaluation metrics

predicted\_labels = torch.argmax(outputs, dim=1)

target\_masks = (labels == 1)

iou = intersection\_over\_union(predicted\_labels, target\_masks)

sensitivity\_val = sensitivity(predicted\_labels, target\_masks)

iou\_list.append(iou)

sensitivity\_list.append(sensitivity\_val)

# Compute average metrics

avg\_iou = torch.mean(torch.cat(iou\_list))

avg\_sensitivity = torch.mean(torch.cat(sensitivity\_list))

estimate\_deviation = 100 - avg\_iou.item()

std\_deviation = torch.std(torch.cat(iou\_list))

print(f"Epoch {epoch+1}: Train Loss: {train\_loss:.4f}, Train Accuracy: {train\_accuracy:.2f}%, IoU: {avg\_iou.item():.2f}, Sensitivity: {avg\_sensitivity.item():.2f}, Estimate Deviation: {estimate\_deviation:.2f}, Standard Deviation: {std\_deviation.item():.2f}")