In [1]:

```
'''
This version use only 10% of the datset to train the model to save time'''
```

In [ ]:

```
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'TRUE'

import numpy as np
from PIL import Image
from torch.utils.data import Dataset
import torch
import torch.nn as nn
import torch.nn.functional as F
from pathlib import Path
import matplotlib.pyplot as plt
```

In [2]:
```
class CustomCloudDataset(Dataset):
    def __init__(self, red_dir, green_dir, blue_dir, nir_dir, gt_dir, transform=None
        self.red_files = list(Path(red_dir).glob('*.tif'))
        self.green_dir = green_dir
        self.blue_dir = blue_dir
        self.nir_dir = nir_dir
        self.gt_dir = gt_dir
        self.transform = transform

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

    def __getitem__(self, idx):
        file_name = self.red_files[idx].name

        red = Image.open(self.red_files[idx])
        green = Image.open(Path(self.green_dir) / file_name.replace('red', 'green'))
        blue = Image.open(Path(self.blue_dir) / file_name.replace('red', 'blue'))
        nir = Image.open(Path(self.nir_dir) / file_name.replace('red', 'nir'))
        gt = Image.open(Path(self.gt_dir) / file_name.replace('red', 'gt'))

        image = np.stack([np.array(red), np.array(green), np.array(blue), np.array(n
        mask = np.array(gt).astype(np.uint8)
        mask = np.where(mask == 255, 1, 0).astype(np.uint8)

        if self.transform:
            image = self.transform(image)
            mask = self.transform(mask)

        image = torch.from_numpy(image).float().permute(2, 0, 1)
        mask = torch.from_numpy(mask).long()

        return image, mask

    def open_as_array(self, idx, include_nir=True):
        file_name = self.red_files[idx].name

        red = np.array(Image.open(self.red_files[idx]))
        green = np.array(Image.open(Path(self.green_dir) / file_name.replace('red', '
        blue = np.array(Image.open(Path(self.blue_dir) / file_name.replace('red', 'b

        if include_nir:
            nir = np.array(Image.open(Path(self.nir_dir) / file_name.replace('red', '
            image = np.stack([red, green, blue, nir], axis=2)
        else:
            image = np.stack([red, green, blue], axis=2)

        # Normalize the image
        max_val = np.iinfo(image.dtype).max
        image = image.astype(np.float32) / max_val

        return image

    def open_mask(self, idx):
        file_name = self.red_files[idx].name
        mask = np.array(Image.open(Path(self.gt_dir) / file_name.replace('red', 'gt')
        mask = np.where(mask == 255, 1, 0)
        return mask
```

In [3]:
```
path = "./38-Cloud_training/"
red_dir = path + "train_red"
green_dir = path + "train_green"
blue_dir = path + "train_blue"
nir_dir = path + "train_nir"
gt_dir = path + "train_gt"

dataset = CustomCloudDataset(red_dir, green_dir, blue_dir, nir_dir, gt_dir)
print(len(dataset))
```

```
8400
```

In [4]:
```
x, y = dataset[1200]
x.shape, y.shape
```

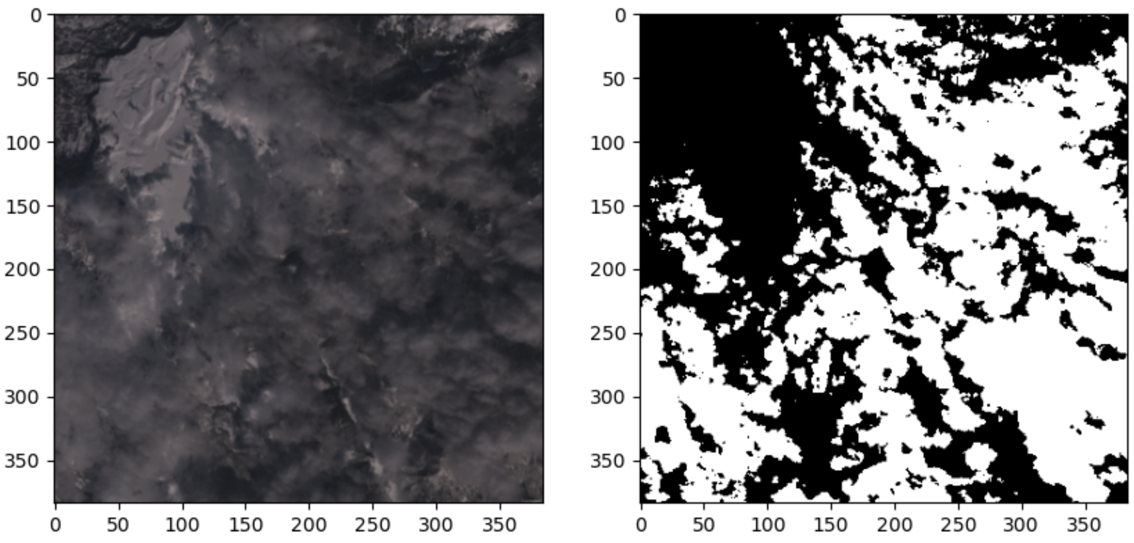
Out[4]:
```
(torch.Size([4, 384, 384]), torch.Size([384, 384]))
```

In [5]:
```
'''
The ideas for the drawing portion were inspired by this article:
https://www.kaggle.com/code/cordmaur/38-cloud-data-preparation?scriptVersionId=31357
'''


fig, ax = plt.subplots(1, 2, figsize=(10, 9))
ax[0].imshow(dataset.open_as_array(1000, include_nir=False))
ax[1].imshow(dataset.open_mask(1000), cmap='gray')

plt.show()
```

```
Image range before plotting: 0.09259174764156342 - 0.6979324221611023
```



In [6]:

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


total_size = len(dataset)
#train_size = int(0.7 * total_size)
#val_size = int(0.15 * total_size)
train_size = int(0.1 * total_size)
val_size = int(0.01 * total_size)
test_size = total_size - train_size - val_size

# Split the dataset
train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size, val_siz

# Create DataLoaders for each set
batch_size = 4
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Check if GPU is available and set the device accordingly
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#device = torch.device("cpu")

print(f"Using device: {device}")
```

```
Using device: cuda
```

In [8]:

```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ConvBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)

    def forward(self, x):
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.relu(self.bn2(self.conv2(x)))
        return x

class UpBlock(nn.Module):
    def __init__(self, in_channels, middle_channels, out_channels):
        super(UpBlock, self).__init__()
        self.up = nn.ConvTranspose2d(in_channels, middle_channels, kernel_size=2, st
        self.conv_block = ConvBlock(middle_channels + out_channels, out_channels)

    def forward(self, x, skip):
        x = self.up(x)
        x = torch.cat([x, skip], dim=1)
        return self.conv_block(x)

class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()
        # Downsampling
        self.down_conv1 = ConvBlock(in_channels, 64)
        self.down_conv2 = ConvBlock(64, 128)
        self.down_conv3 = ConvBlock(128, 256)
        self.down_conv4 = ConvBlock(256, 512)

        # Upsampling
        self.up_block1 = UpBlock(512, 256, 256)  # Input 512, middle 256, concatenate
        self.up_block2 = UpBlock(256, 128, 128)  # Input 256, middle 128, concatenate
        self.up_block3 = UpBlock(128, 64, 64)    # Input 128, middle 64, concatenate

        # Final layer
        self.final_conv = nn.Conv2d(64, out_channels, kernel_size=1)

    def forward(self, x):
        # Downsampling
        x1 = self.down_conv1(x)
        x2 = self.down_conv2(F.max_pool2d(x1, 2))
        x3 = self.down_conv3(F.max_pool2d(x2, 2))
        x4 = self.down_conv4(F.max_pool2d(x3, 2))

        # Upsampling
        x = self.up_block1(x4, x3)
        x = self.up_block2(x, x2)
        x = self.up_block3(x, x1)

        # Final layer
        out = self.final_conv(x)
        return out
```

In [9]:

```
n_channels = 4 # Number of channels in the input image (e.g., RGB+NIR)

# The number of output channels is 2 for binary classification
model = UNet(n_channels, 2)

loss_fn = nn.CrossEntropyLoss()
```

In [10]:

```
print(model)
```

```
UNet(
  (down_conv1): ConvBlock(
    (conv1): Conv2d(4, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_sta
ts=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_sta
ts=True)
  )
  (down_conv2): ConvBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
  )
  (down_conv3): ConvBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
  )
  (down_conv4): ConvBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
  )
  (up_block1): UpBlock(
    (up): ConvTranspose2d(512, 256, kernel_size=(2, 2), stride=(2, 2))
    (conv_block): ConvBlock(
      (conv1): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    )
  )
  (up_block2): UpBlock(
    (up): ConvTranspose2d(256, 128, kernel_size=(2, 2), stride=(2, 2))
    (conv_block): ConvBlock(
      (conv1): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    )
  )
  (up_block3): UpBlock(
    (up): ConvTranspose2d(128, 64, kernel_size=(2, 2), stride=(2, 2))
    (conv_block): ConvBlock(
      (conv1): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    )
  )
  (final_conv): Conv2d(64, 2, kernel_size=(1, 1), stride=(1, 1))
)
```

```
In [12]: def train_model(model, train_dl, valid_dl, loss_fn, optimizer, epochs, n_classes):

             device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
             model.to(device)

             for epoch in range(epochs):
                 print(f'Epoch {epoch}/{epochs - 1}')
                 print('-' * 10)

                 # Training phase
                 model.train()
                 train_loss, train_corrects, train_pixels = 0, 0, 0
                 for inputs, labels in train_loader:
                     inputs, labels = inputs.to(device), labels.to(device)
                     outputs = model(inputs)

                     # Debugging purpose
                     if torch.any(labels >= n_classes):
                         print(f"Invalid label detected. Labels should be less than {n_classes

                     optimizer.zero_grad()

                     outputs = model(inputs)
                     loss = loss_fn(outputs, labels)

                     loss.backward()
                     optimizer.step()

                     # Convert the model's output to predicted classes
                     preds = outputs.argmax(dim=1)

                     # Update training metrics
                     train_loss += loss.item() * inputs.size(0)
                     train_corrects += torch.sum(preds == labels.data)
                     train_pixels += torch.numel(labels)

                 # Calculate and print training loss and accuracy
                 train_epoch_loss = train_loss / len(train_dl.dataset)
                 train_epoch_acc = train_corrects.double() / train_pixels
                 print(f'Train Loss: {train_epoch_loss:.4f} Acc: {train_epoch_acc:.4f}')

                 # Validation phase
                 model.eval()
                 valid_loss, valid_corrects, valid_pixels = 0, 0, 0
                 with torch.no_grad():
                     for inputs, labels in valid_dl:
                         inputs, labels = inputs.to(device), labels.to(device)

                         outputs = model(inputs)
                         loss = loss_fn(outputs, labels)

                         preds = outputs.argmax(dim=1)
                         valid_loss += loss.item() * inputs.size(0)
                         valid_corrects += torch.sum(preds == labels.data)
                         valid_pixels += torch.numel(labels)

                 valid_epoch_loss = valid_loss / len(valid_dl.dataset)
                 valid_epoch_acc = valid_corrects.double() / valid_pixels
                 print(f'Valid Loss: {valid_epoch_loss:.4f} Acc: {valid_epoch_acc:.4f}')
```

```
    print('Training complete')
```

In [13]:

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

```
WARNING:tensorflow:From C:\Users\HP\anaconda3\Lib\site-packages\keras\src\losses.p
y:2976: The name tf.losses.sparse_softmax_cross_entropy is deprecated. Please use
tf.compat.v1.losses.sparse_softmax_cross_entropy instead.

WARNING:tensorflow:From C:\Users\HP\anaconda3\Lib\site-packages\keras\src\backend.
py:873: The name tf.get_default_graph is deprecated. Please use tf.compat.v1.get_d
efault_graph instead.
```

In [14]:

```
epochs = 10  # Set the number of epochs
n_classes = 2
train_model(model, train_loader, val_loader, loss_fn, optimizer, epochs, n_classes)
```

```
Epoch 0/9
----------
Train Loss: 0.4283 Acc: 0.7762
Valid Loss: 0.3526 Acc: 0.8010
Epoch 1/9
----------
Train Loss: 0.4152 Acc: 0.8008
Valid Loss: 0.3734 Acc: 0.7791
Epoch 2/9
----------
Train Loss: 0.3767 Acc: 0.8038
Valid Loss: 0.3157 Acc: 0.8246
Epoch 3/9
----------
Train Loss: 0.3901 Acc: 0.7960
Valid Loss: 0.3329 Acc: 0.8385
Epoch 4/9
----------
Train Loss: 0.3815 Acc: 0.8053
Valid Loss: 0.3159 Acc: 0.8508
Epoch 5/9
----------
Train Loss: 0.3756 Acc: 0.8101
Valid Loss: 0.3548 Acc: 0.7844
Epoch 6/9
----------
Train Loss: 0.4042 Acc: 0.8028
Valid Loss: 0.3233 Acc: 0.8511
Epoch 7/9
----------
Train Loss: 0.3698 Acc: 0.8279
Valid Loss: 0.3066 Acc: 0.8452
Epoch 8/9
----------
Train Loss: 0.3784 Acc: 0.8270
Valid Loss: 0.2845 Acc: 0.8697
Epoch 9/9
----------
Train Loss: 0.3887 Acc: 0.8120
Valid Loss: 0.2866 Acc: 0.8509
Training complete
```

In [ ]:

```
```