Question: How would you find the boundary of the diffusion of each dye? (Note: If you have multiple answers in mind, break them apart and explain each one separately.) Explain each solution/algorithm in detail.

Answers: The following steps are used to find diffusion boundaries of each dyes:

1. Extract color channels corresponding to each dyes (Red: channel 0, Green: channel 1).
2. Convert single-channel images to binary images using thresholding (Red: 0.06, Green: 0.06)
3. Connect neighboring pixels by filling small holes less than a certain pixels (Red: 64 pixels, Green: 128 pixels).
4. Remove noises by eliminating connected components that are less than a certain threshold (Red: 20000 pixels, Green: 64 pixels).
5. Soften edges by dilating the image with a certain matrix (Red: 3x3, Green: 4x4).
6. Find edges using Roberts Cross filter.

```python
from skimage.io import imread
import matplotlib.pyplot as plt
from skimage.filters import roberts
from skimage.morphology import (
    dilation,
    square,
    remove_small_objects,
    remove_small_holes,
)
import numpy as np
from pathlib import Path

ROOT = Path(__file__).parent

def find_red_diffusion_boundary(img: np.ndarray) -> np.ndarray:
    """Find the red diffusion boundary.

    Args:
        img (np.ndarray): images.

    Returns:
        np.ndarray: binary images where 1s indicate the boundary.
    """
    # Extract the red channel from the image and normalize it.
    img = img[:, :, 0] / 255.0

    # Convert the image to binary image with thresholding.
    img = img > 0.06

    #  Connected neighboring pixels by filling small holes that are less than 64 pixels.
    img = remove_small_holes(img, 64)

    # Reduce noises by eliminating connected components that are less than 20,000 pixels.
    img = remove_small_objects(img, 20000)

    # Soften edges by dilating the image with 3x3 matrix
    img = dilation(img, square(3))

    # Fill in the remaining holes
    img = remove_small_holes(img, 128)

    # Use roberts filter to find edges
    img = roberts(img)
    img = img != 0

    return img


def find_green_diffusion_boundary(img: np.ndarray) -> np.ndarray:
    """Find the green diffusion boundary.

    Args:
        img (np.ndarray): images.

    Returns:
        np.ndarray: binary images where 1s indicate the boundary.
    """
```

```python
    # Extract the green channel from the image and normalize it.
    img = img[:, :, 1] / 255.0

    # Convert the image to binary image with thresholding.
    img = img > 0.06

    # Connected neighboring pixels by filling small holes that are less than 128 pixels.
    img = remove_small_holes(img, 128)

    # Reduce noises by eliminating connected components that are less than 64 pixels.
    img = remove_small_objects(img, 64)

    # Soften edges by dilating the image with 4x4 matrix
    img = dilation(img, square(4))

    # Fill in the remaining holes
    img = remove_small_holes(img, 1024)

    # Use roberts filter to find edges
    img = roberts(img)
    img = img != 0

    return img


def problem_1(in_img: str = ROOT/"dyes.png", out_img: str =ROOT/"problem_1_result.png") -> None:
    """Find diffusion boundaries of the red and the green pixels.

    Args:
        in_img (str, optional): path to the input image. Defaults to "./dyes.png".
        out_img (str, optional): path to the output image. Defaults to "./problem1_reuslt.png".
    """
    # Load the image
    img = imread(in_img)

    # Find boundaries
    red_diffusion_boundary = find_red_diffusion_boundary(img)
    green_diffusion_boundary = find_green_diffusion_boundary(img)

    # Find boundaries' coordinates
    red_coordinates = np.where(red_diffusion_boundary == 1)
    green_coordinates = np.where(green_diffusion_boundary == 1)

    # Plot boundaries
    img[red_coordinates[0], red_coordinates[1]] = [255, 0, 0]
    img[green_coordinates[0], green_coordinates[1]] = [0, 255, 0]

    # Save the resulting image
    plt.imsave(out_img, img)


if __name__ == "__main__":
    problem_1()
```

Question: How would you create your model to classify the digits? (Note: If you have multiple answers in mind, break them apart and explain each one separately.) Explain each solution/algorithm in detail.

Answers: The following steps are used to create a model that is able to recognize handwritten digits.

1. Load data and labels from mat files (data: 784x2500, labels: 1x2500).
2. Reshape data and labels (data:2500x784, labels:2500)
3. Perform K-Fold Cross Validation to split the dataset into 15 folds.
4. For each fold,...
    - 4.1. Split the train dataset into a train dataset and a validation dataset. Keep the test dataset the same.
    - 4.2. Load all three datasets into Dataset classes to convert all values to Tensor.
    - 4.3. Load train_dataset into a Dataloader class to perform batch training.
    - 4.4. Define the model, the loss function, and the optimizer.
    - 4.5. At each epoch, ...
        - 4.5.1. Feed data from train_dataloader into the model to produce predicted labels.
        - 4.5.2. Use the loss function to calculate a loss between predicted labels and truth labels.
        - 4.5.3. Calculate gradients of each model's weights with respect to the loss.
        - 4.5.4. Use the optimizer to update a model's weights.
        - 4.5.5. Calculate train_loss and train_accuracy.
        - 4.5.6. Validate the model with val_dataset and calculate val_loss and val_accuracy.
        - 4.5.7. If val_loss has not decreased in the last 10 epochs, end the training process.
        - 4.5.8. Else, train the model until the final epoch.
    - 4.6. Test the model with test_dataset and calculate test_loss and test_accuracy.
5. Calculate the average loss and the average accuracy for all three datasets across all folds.

Design details:

- Model: Fully-connected Model

```
Given: in_features, num_classes
Model(
    Linear(in_features=in_features, out_features=32)
    ReLU()
    Linear(in_features=32, out_features=32)
    ReLU()
    Linear(in_features=32, out_features=num_classes)
)
```

Ans: As shown above, the model that is used to recognize handwritten digits is a fully connected model composed of three hidden layers. As with any fully-connected model, each node in a layer is connected to all nodes in the next layer. ReLU is the activation function for the first and the second hidden layers and it serves to introduce nonlinearity into the model.

- Loss function: Cross-Entropy Loss

Ans: Cross-Entropy is the loss function of choice as it is designed to measure differences between classes. The implementation of Cross-Entropy loss in PyTorch utilizes the sigmoid function which ensures all values fall between 0 and 1 and the negative log loss where loss goes to 0 as input approaches one.

- Optimizer: Adam

Ans: Adam is an adaptive optimizer that works well in most cases. It utilizes an adaptive learning rate where each weight has its own learning rate and momentum where learning rates get adjusted based on the gradient direction and time.

```python
from pathlib import Path
from scipy.io import loadmat
from sklearn.model_selection import KFold
import torch
from torch.nn import CrossEntropyLoss
from torch.optim import Adam
from torch.utils.data import DataLoader
from dataset import CustomDataset
from model import Model
from sklearn.model_selection import train_test_split
from utilities import train, test

ROOT = Path(__file__).parent


def problem_2(
,    data_path: str = ROOT / "digits.mat", labels_path: str = ROOT / "labels.mat"
,):
    """Perform K-Fold validation and use those data to train a neural network. The training process utilizes early stopping to prevent overfitting.

    Args:
        data_path (str, optional): path to the data file. Defaults to ROOT/"digits.mat".
        labels_path (str, optional): path to the labels file. Defaults to ROOT/"labels.mat".
    """
    # Hyperparameters
    n_splits = 15
    lr = 0.001
    batch_size = 32
    num_epochs = 1000
    patient = 10

    # Load data from .mat files
    data = loadmat(data_path)["data"].T
    labels = loadmat(labels_path)["labels"].T.flatten()

    # K-Fold cross validation
    kf = KFold(n_splits, shuffle=True)

    # Keep track of currrent fold
    split = 0

    # Initialize average metrics
    avg_train_loss = 0
    avg_train_acc = 0
    avg_val_loss = 0
    avg_val_acc = 0
    avg_test_loss = 0
    avg_test_acc = 0

    # Perform k-fold cross validation
    for train_indices, test_indices in kf.split(data):
        # Split dataset into train, valid, and test set
        X_train, X_val, y_train, y_val = train_test_split(
            data[train_indices], labels[train_indices]
        )
        X_test, y_test = data[test_indices], labels[test_indices]
```

```python

        # Convert numpy array to dataset
        train_dataset = CustomDataset(X_train, y_train)
        val_dataset = CustomDataset(X_val, y_val)
        test_dataset = CustomDataset(X_test, y_test)

        # Load train dataset into dataloader for batch training
        train_dataloader = DataLoader(train_dataset, batch_size)

        # Define features and classes
        in_shape = torch.from_numpy(data).size()
        num_classes = torch.unique(torch.from_numpy(labels)).size()[0]

        # Define model, loss function, and optimizer
        model = Model(in_shape, num_classes)
        loss_fn = CrossEntropyLoss()
        optimizer = Adam(model.parameters(), lr)

        # Initialize per fold metrices
        train_loss = 0
        train_acc = 0
        val_loss = 0
        val_acc = 0
        test_loss = 0
        test_acc = 0

        # Early Stop: store multiple losses
        es_loss_list = []

        # Train the model
        for epoch in range(num_epochs):

            # Batch training
            train_loss, train_acc, model = train(
                model, loss_fn, optimizer, train_dataloader
            )

            # Validate the model
            val_loss, val_acc, model = test(model, loss_fn, val_dataset)

            # Print per epoch metrices
            print(
                f"Epoch: {epoch+1}/{num_epochs},",
                f" train_loss:{train_loss:.5f},",
                f" train_acc: {train_acc:.3f},",
                f" val_loss:{val_loss:.5f},",
                f" val_acc:{val_acc:.3f}",
                end="\r",
            )

            # Early stop algorithm
            # Add latest loss to the end of the list
            es_loss_list.append(val_loss)

            # Remove first loss if the list is larger than patient
            if len(es_loss_list) > patient:
                es_loss_list = es_loss_list[1:]
```

```python

            # Check if the list is not decreasing
            not_decreasing = all(a <= b for a, b in zip(es_loss_list, es_loss_list[1:]))

            # Stop training once val loss list not decreasing
            if not_decreasing and len(es_loss_list) == patient:
                break

        # Test the model
        test_loss, test_acc, model = test(model, loss_fn, test_dataset)

        # Print per fold metrices
        print(
            f"Fold: {split+1}/{n_splits},",
            f" test_loss:{test_loss:.5f},",
            f" test_acc:{test_acc:.3f},",
            f" train_loss:{train_loss:.5f},",
            f" train_acc: {train_acc:.3f},",
            f" val_loss:{val_loss:.5f},",
            f" val_acc:{val_acc:.3f}",
        )

        # Update average metrices
        avg_val_loss += val_loss / n_splits
        avg_val_acc += val_acc / n_splits
        avg_train_loss += train_loss / n_splits
        avg_train_acc += train_acc / n_splits
        avg_test_loss += test_loss / n_splits
        avg_test_acc += test_acc / n_splits

        # Update split
        split += 1

    # Print average metrices
    print(
        "----Average Metrices----\n",
        f"num_folds: {n_splits}\n",
        f"avg_test_acc: {avg_test_acc:.3f}\n",
        f"avg_train_acc: {avg_train_acc:.3f}\n",
        f"avg_val_acc: {avg_val_acc:.3f}\n",
        f"avg_test_loss: {avg_test_loss:.5f}\n",
        f"avg_train_loss: {avg_train_loss:.5f}\n",
        f"avg_val_loss: {avg_val_loss:.5f}",
    )


if __name__ == "__main__":
    problem_2()
```

```python
import re
import torch
from torch.nn import Module, CrossEntropyLoss
from torch.utils.data import DataLoader, Dataset
from torch.optim import Adam


def train(
,    model: Module, loss_fn: CrossEntropyLoss, optimizer: Adam, dataloader: DataLoader
,) -> tuple:
    """Train a given model.

    Args:
        model (Module): model.
        loss_fn (CrossEntropyLoss): loss function.
        optimizer (Adam): optimizer.
        dataloader (DataLoader): dataloader.

    Returns:
        tuple: loss, accuracy, model.
    """
    # Keep tracking for y and y_pred to calculate final loss and accuracy
    all_y = None
    all_y_pred = None

    # Perform batch training
    for _, (X, y) in enumerate(dataloader):

        # Forward Propagation
        y_pred = model(X)

        # Calculate loss
        loss = loss_fn(y_pred, y)

        # Calculate gradients
        loss.backward()

        # Update weights
        optimizer.step()

        # Clearn gradients in the optimizer
        optimizer.zero_grad()

        # Store y and y_pred of this batch
        with torch.no_grad():
            all_y = y if all_y == None else torch.cat((all_y, y))
            all_y_pred = (
                y_pred if all_y_pred == None else torch.cat((all_y_pred, y_pred))
            )

    # Calculate loss and accuracy
    loss, acc = calculate_loss_accuracy(all_y_pred, all_y, loss_fn)

    return loss, acc, model


def test(model: Module, loss_fn: CrossEntropyLoss, dataset: Dataset) -> tuple:
```

```
    """Test a given model.

    Args:
        model (Module): model.
        loss_fn (CrossEntropyLoss): loss function.
        dataset (Dataset): dataset.

    Returns:
        tuple: loss, accuracy, model
    """
    # Use no_grad to freeze the model.
    with torch.no_grad():
        X = dataset.X
        y = dataset.y

        # Forward Propagation
        y_pred = model(X)

        # Calculate loss and accuracy
        loss, acc = calculate_loss_accuracy(y_pred, y, loss_fn)

    return loss, acc, model


def calculate_loss_accuracy(
,    y_pred: torch.Tensor, y: torch.tensor, loss_fn: CrossEntropyLoss
,) -> tuple:
    """Calculate loss and accuracy with given labels and predicted labels.

    Args:
        y_pred (torch.Tensor): predicted labels.
        y (torch.tensor): true labels.
        loss_fn (CrossEntropyLoss): loss function.

    Returns:
        tuple: loss, accuracy
    """
    acc = torch.sum(torch.argmax(y_pred, 1) == y) / y.size()[0]
    loss = loss_fn(y_pred, y).item()
    return loss, acc
```

```python
import torch
from torch.nn import Module, Linear
from torch.nn.functional import relu


class Model(Module):
    """An implementation of torch.nn.Module.

    Args:
        Module (Class): generic pytroch model class.
    """

    def __init__(self, in_shape: torch.Size, num_classes: int):
        """Initialize the model

        Args:
            in_shape (torch.Size): the shape of input.
            num_classes (int, optional): number of output classes.
        """
        super(Model, self).__init__()

        # Parameters
        self.in_features = torch.prod(torch.tensor(in_shape[1:]))
        self.num_classes = num_classes

        # Define layers
        self.fc0 = Linear(self.in_features, 32)
        self.fc1 = Linear(32, 32)
        self.fc2 = Linear(32, self.num_classes)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Feed data through the model.

        Args:
            x (torch.Tensor): data.

        Returns:
            torch.Tensor: label.
        """
        x = relu(self.fc0(x))
        x = relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```python
from torch.utils.data import Dataset
import torch
import numpy as np


class CustomDataset(Dataset):
    """An implementation of torch.utils.data.Dataset .

    Args:
        Dataset (Class): generic pytorch dataset class.
    """

    def __init__(self, X: np.ndarray, y: np.ndarray):
        """Initialize the dataset

        Args:
            X (np.ndarray): data.
            y (np.ndarray): labels.
        """
        super(CustomDataset, self).__init__()
        self.X = torch.from_numpy(X).float()
        self.y = torch.from_numpy(y)

    def __getitem__(self, index: int) -> tuple:
        """Return data and label based on index.

        Args:
            index (int): index.

        Returns:
            tuple: data, label
        """
        return self.X[index], self.y[index]

    def __len__(self) -> int:
        """Return dataset length

        Returns:
            int: length
        """
        return self.X.shape[0]
```