# StrongLensingChallenge_Classification_Task_A

July 29, 2025

## 1 Task A: Multi-Class Classification

Gravitational lensing has been a cornerstone in many cosmology experiments and studies since it was discussed in Einstein's calculations back in 1936 and discovered in 1979, and one area of particular interest is the study of dark matter via substructure in strong lensing images. In this challenge, we focus on exploring the potential of supervised models in identifying dark matter based on simulated strong lensing images with different substructure.

This is an example notebook for the Multi-Class Classification Challenge. In this notebook, we demonstrate a simple CNN model implemented using the PyTorch library to solve the task of multi-class classification of strong lensing images.

#### 1.0.1 Dataset

The Dataset consists of three classes, strong lensing images with no substructure, CDM (cold dark matter) substructure, and axion substructure. The images have been normalized using min-max normalization, but you are free to use any normalization or data augmentation methods to improve your results.

Link to the Dataset: https://drive.google.com/file/d/1AZAJzJdm6FJT4rIyY9N_6FcKLf8VZtn8/view?usp=sharin

#### 1.0.2 Evaluation Metrics

- ROC curve (Receiver Operating Characteristic curve) and AUC score (Area Under the ROC Curve)

- Accuracy, Precision, Recall, and F1 Score

The model performance will be tested on the hidden test dataset based on the above metrics. More details about these metrics and the code to calculate them has been shared below.

#### 1.0.3 Instructions for using the notebook

1. Use GPU acceleration: (Edit –> Notebook settings –> Hardware accelerator –> GPU)
2. Run the cells: (Runtime –> Run all)

[14]:
```
!pip install gdown
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: gdown in /home/diogo/.local/lib/python3.10/site-
packages (5.2.0)
```

```
Requirement already satisfied: tqdm in /home/diogo/.local/lib/python3.10/site-
packages (from gdown) (4.67.1)
Requirement already satisfied: beautifulsoup4 in
/home/diogo/.local/lib/python3.10/site-packages (from gdown) (4.13.4)
Requirement already satisfied: requests[socks] in
/home/diogo/.local/lib/python3.10/site-packages (from gdown) (2.32.4)
Requirement already satisfied: filelock in
/home/diogo/.local/lib/python3.10/site-packages (from gdown) (3.18.0)
Requirement already satisfied: typing-extensions>=4.0.0 in
/home/diogo/.local/lib/python3.10/site-packages (from beautifulsoup4->gdown)
(4.14.1)
Requirement already satisfied: soupsieve>1.2 in
/home/diogo/.local/lib/python3.10/site-packages (from beautifulsoup4->gdown)
(2.7)
Requirement already satisfied: charset_normalizer<4,>=2 in
/home/diogo/.local/lib/python3.10/site-packages (from requests[socks]->gdown)
(3.4.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/lib/python3/dist-packages
(from requests[socks]->gdown) (3.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/lib/python3/dist-
packages (from requests[socks]->gdown) (2020.6.20)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/lib/python3/dist-
packages (from requests[socks]->gdown) (1.26.5)
Requirement already satisfied: PySocks!=1.5.7,>=1.5.6 in
/home/diogo/.local/lib/python3.10/site-packages (from requests[socks]->gdown)
(1.7.1)
```

[15]:
```python
import gdown
```

[16]:
```python
import os
# Check if the dataset folder is missing
if not os.path.exists('./dataset'):
    # Download and extract the dataset
    !gdown "http://drive.google.com/uc?id=1AZAJzJdm6FJT4rIyY9N_6FcKLf8VZtn8"
    !unzip -q dataset.zip
```

## 1.1 Multi-Class Classification using a Supervised Model

### 1.1.1 1. Data Visualization and Preprocessing

**1.1 Import all the necessary libraries**

[ ]:
```python
import numpy as np
import random
import matplotlib.pyplot as plt
import torch
import torchvision
import torch.nn as nn
import torch.utils.data as data
```

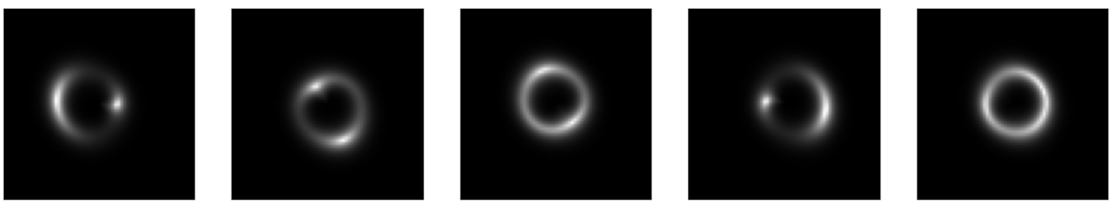
```
from torchvision.datasets import DatasetFolder
from numpy import interp
from itertools import cycle
from tqdm.notebook import tqdm
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score
from sklearn.preprocessing import label_binarize
import torch.utils.model_zoo as model_zoo
%matplotlib inline
```

### 1.2 Preview the Data

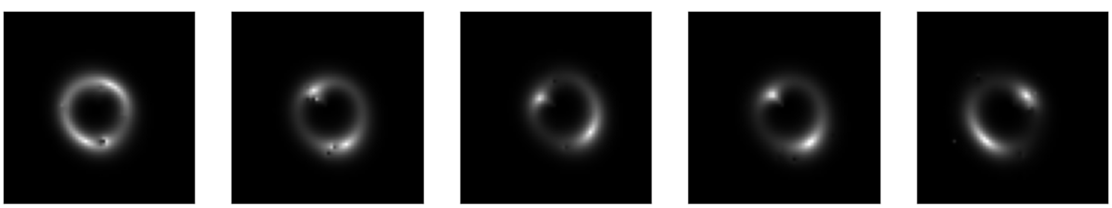
[18]:
```
# Define the input paths
train_path1 = './dataset/train/no'
train_files1 = [os.path.join(train_path1, f) for f in os.listdir(train_path1)␣
↪if f.endswith(".npy")]
train_path2 = './dataset/train/cdm'
train_files2 = [os.path.join(train_path2, f) for f in os.listdir(train_path2)␣
↪if f.endswith(".npy")]
train_path3 = './dataset/train/axion'
train_files3 = [os.path.join(train_path3, f) for f in os.listdir(train_path3)␣
↪if f.endswith(".npy")]

# Number of samples to display per class
n = 5

# Plot the samples with no substructure
i = 1
print('Samples with no substructure: ')
plt.rcParams['figure.figsize'] = [14, 14]  # Set the figure size
for image in train_files1[:n]:
    ax = plt.subplot(3, n, i)  # Create subplot
    plt.imshow(np.load(image).reshape(64, 64), cmap='gray')  # Load and display␣
↪the image
    ax.get_xaxis().set_visible(False)  # Hide x-axis
    ax.get_yaxis().set_visible(False)  # Hide y-axis
    i += 1
plt.show()  # Show the plot

# Plot the samples with spherical substructure
print('Samples with spherical substructure: ')
plt.rcParams['figure.figsize'] = [14, 14]  # Set the figure size
for image in train_files2[:n]:
    ax = plt.subplot(3, n, i)  # Create subplot
    plt.imshow(np.load(image).reshape(64, 64), cmap='gray')  # Load and display␣
↪the image
```

```
    ax.get_xaxis().set_visible(False)  # Hide x-axis
    ax.get_yaxis().set_visible(False)  # Hide y-axis
    i += 1
plt.show()  # Show the plot

# Plot the samples with vortex substructure
print('Samples with vortex substructure: ')
plt.rcParams['figure.figsize'] = [14, 14]  # Set the figure size
for image in train_files3[:n]:
    ax = plt.subplot(3, n, i)  # Create subplot
    plt.imshow(np.load(image).reshape(64, 64), cmap='gray')  # Load and display the image
    ax.get_xaxis().set_visible(False)  # Hide x-axis
    ax.get_yaxis().set_visible(False)  # Hide y-axis
    i += 1
```

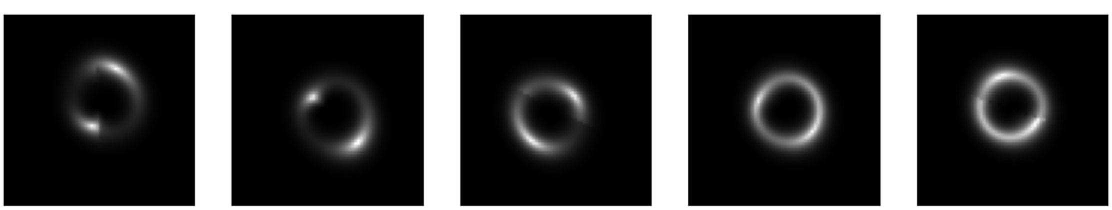
```
Samples with no substructure:
```



```
Samples with spherical substructure:
```



```
Samples with vortex substructure:
```



### 1.3 Add Gausian noise

[ ]:
```
class NoisyDataset(torch.utils.data.Dataset):
    def __init__(self, dataset, noise_probability = 0.1, noise_std = 0.1):
        self.dataset = dataset
        self.noise_probability = noise_probability
        self.noise_std = noise_std

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

    def __getitem__(self, index):
        sample, label = self.dataset[index]

        if random.random() < self.noise_probability:
            noise = torch.randn_like(sample) * self.noise_std
            sample = sample + noise
            sample = torch.clamp(sample, 0.0, 1.0)

        return sample, label
```

### 1.4 Import Training and Validation Data

[20]:
```
# Set Batch Size
batch_size = 100

# Define a function to load .npy files
def npy_loader(path):
    sample = torch.from_numpy(np.load(path))  # Load the numpy file and convert␣
 ↪it to a torch tensor
    return sample

# Load training data
train_data = torchvision.datasets.DatasetFolder(root='./dataset/train',␣
 ↪loader=npy_loader, extensions='.npy')
print("Training Classes: " + str(train_data.class_to_idx))  # Print the classes␣
 ↪found in the training data
train_data_loader = data.DataLoader(train_data, batch_size=batch_size,␣
 ↪shuffle=True, num_workers=4)  # Create a data loader for training data

train_data = NoisyDataset(train_data, noise_probability=0.1, noise_std=0.1)

# Load validation data
val_data = torchvision.datasets.DatasetFolder(root='./dataset/val',␣
 ↪loader=npy_loader, extensions='.npy')
print("Validation Classes: " + str(val_data.class_to_idx))  # Print the classes␣
 ↪found in the validation data
```

```
val_data_loader = data.DataLoader(val_data, batch_size=batch_size,␣
↪shuffle=True, num_workers=4)  # Create a data loader for validation data
```

```
Training Classes: {'axion': 0, 'cdm': 1, 'no': 2}
Validation Classes: {'axion': 0, 'cdm': 1, 'no': 2}
```

#### 1.1.2 2. Training

**2.1 Defining a ResNet + CBAM Model**

[ ]:
```
model_urls = {
    'resnet18': 'https://download.pytorch.org/models/resnet18-5c106cde.pth',
}



def conv3x3(in_planes, out_planes, stride=1):
    "3x3 convolution with padding"
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
                     padding=1, bias=False)

class ChannelAttention(nn.Module):
    def __init__(self, in_planes, ratio=16):
        super(ChannelAttention, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.max_pool = nn.AdaptiveMaxPool2d(1)

        self.fc = nn.Sequential(nn.Conv2d(in_planes, in_planes // 16, 1,␣
↪bias=False),
                               nn.ReLU(),
                               nn.Conv2d(in_planes // 16, in_planes, 1,␣
↪bias=False))
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avg_out = self.fc(self.avg_pool(x))
        max_out = self.fc(self.max_pool(x))
        out = avg_out + max_out
        return self.sigmoid(out)

class SpatialAttention(nn.Module):
    def __init__(self, kernel_size=7):
        super(SpatialAttention, self).__init__()

        self.conv1 = nn.Conv2d(2, 1, kernel_size, padding=kernel_size//2,␣
↪bias=False)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
```

```
        avg_out = torch.mean(x, dim=1, keepdim=True)
        max_out, _ = torch.max(x, dim=1, keepdim=True)
        x = torch.cat([avg_out, max_out], dim=1)
        x = self.conv1(x)
        return self.sigmoid(x)

class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)

        self.ca = ChannelAttention(planes)
        self.sa = SpatialAttention()

        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.ca(out) * out
        out = self.sa(out) * out
        out = self.relu(out)

        return out


class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=1000, grayscale=False):
        self.inplanes = 64
```

```python
        super(ResNet, self).__init__()

        # Handle grayscale input
        in_channels = 1 if grayscale else 3
        self.conv1 = nn.Conv2d(in_channels, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)

        # Initialize weights
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, planes * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion),
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample))
        self.inplanes = planes * block.expansion
        for _ in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
```

```
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x

def resnet18_cbam(pretrained=False, grayscale=False, num_classes=1000):
    """Constructs a ResNet-18 model with CBAM attention.

    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        grayscale (bool): If True, expects 1-channel input instead of 3-channel
        num_classes (int): Number of output classes
    """
    model = ResNet(BasicBlock, [2, 2, 2, 2], num_classes=num_classes, grayscale=grayscale)

    if pretrained:
        pretrained_dict = model_zoo.load_url(model_urls['resnet18'])
        model_dict = model.state_dict()

        # 1. Filter out keys not present in the model or incompatible
        pretrained_dict = {
            k: v for k, v in pretrained_dict.items()
            if (k in model_dict)
            and (not grayscale or not k.startswith('conv1.'))
            and ('ca.' not in k)
            and ('sa.' not in k)
            and not k.startswith('fc.')
        }

        # 2. Update the model's state dict
        model_dict.update(pretrained_dict)

        # 3. Load the modified state dict
        model.load_state_dict(model_dict)

        # 4. Initialize CBAM layers
        for m in model.modules():
```

```
            if isinstance(m, ChannelAttention):
                nn.init.kaiming_normal_(m.fc[0].weight, mode='fan_out')
                nn.init.kaiming_normal_(m.fc[2].weight, mode='fan_out')
            elif isinstance(m, SpatialAttention):
                nn.init.kaiming_normal_(m.conv1.weight, mode='fan_out')

    return model

# Set the device to GPU if available, otherwise use CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Instantiate the model and move it to the appropriate device
model = resnet18_cbam(pretrained=True, grayscale=True, num_classes=3).to(device)
```

### 2.2 Training the ResNet + CBAM Model

[23]:
```
# Loss Function
criteria = nn.CrossEntropyLoss()

# Optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-5)

# Calculate the number of batches for training data
n_batches_train = (len(train_files1) * 3) / batch_size  # Equal number of files␣
↪in each class

# Set the number of training epochs
n_epochs = 50
loss_array = []  # To store the loss values

# Progress bar for epochs
pbar = tqdm(range(1, n_epochs + 1))
for epoch in pbar:
    train_loss = 0.0
    train_acc = 0.0

    # Iterate over the training data loader
    for step, (x_tr, y_tr) in enumerate(train_data_loader):
        data = x_tr.to(device).float()  # Move input data to the device and␣
↪convert to float
        labels = y_tr.to(device, dtype=torch.long)  # Move labels to the device␣
↪and convert to long
        optimizer.zero_grad()  # Clear the gradients
        outputs = model(data)  # Forward pass through the model
        _, preds = torch.max(outputs.data, 1)  # Get the predictions
        correct = (preds == labels).float().sum()  # Calculate the number of␣
↪correct predictions
```

```
        loss = criteria(outputs, labels)  # Calculate the loss
        loss.backward()  # Backpropagation
        optimizer.step()  # Update the model parameters

        train_loss += loss.item()  # Accumulate the loss
        train_acc += correct.item() / data.shape[0]  # Accumulate the accuracy

    # Calculate the average loss and accuracy for the epoch
    train_loss = train_loss / n_batches_train
    train_acc = train_acc / n_batches_train

    # Display the training statistics
    pbar.set_postfix({'Training Loss': train_loss, 'Training Acc': train_acc})
```

```
  0%|          | 0/50 [00:00<?, ?it/s]
```

### 1.1.3 3. Testing

**3.1 Testing the CNN Model on Validation Data**

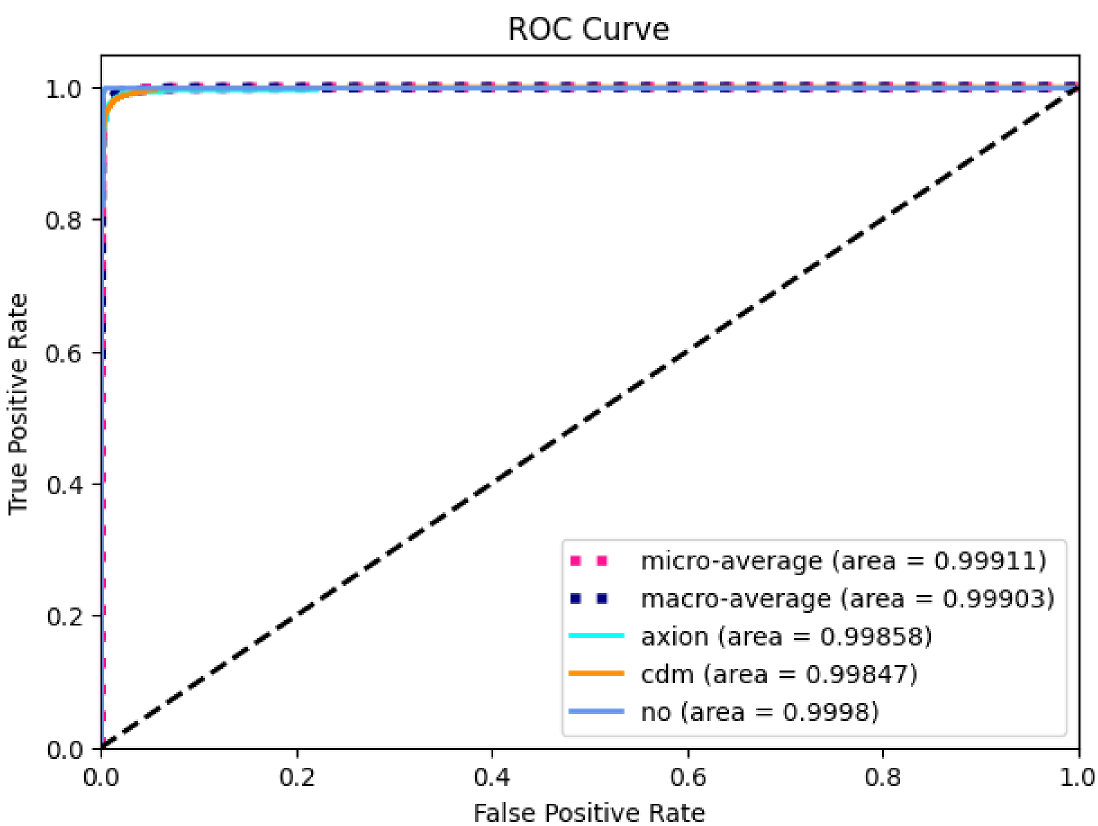
[24]:
```
# Initialize lists to store scores and labels
y_score = []
y_test = []

# Iterate over the validation data loader
for _, (x_ts, y_ts) in enumerate(val_data_loader):
    mini_val_data = x_ts.to(device).float()  # Move validation data to the␣
 ↪device and convert to float
    y_ts = y_ts.to(device, dtype=torch.long)  # Move labels to the device and␣
 ↪convert to long

    with torch.no_grad():  # Disable gradient calculation for validation
        outputs = model(mini_val_data)  # Forward pass through the model
        probabilities = torch.nn.functional.softmax(outputs, dim=1)  # Apply␣
 ↪softmax to get probabilities

    # Append the probabilities and labels to the respective lists
    y_score.append(probabilities.cpu().detach().numpy())
    y_test.append(y_ts.cpu().detach().numpy())

# Convert the lists to numpy arrays and reshape them
y_score = np.asarray(y_score).reshape(-1, 3)
y_val = np.asarray(y_test).reshape(-1)

# Binarize the labels for multi-class evaluation
y_val = label_binarize(y_val, classes=[0, 1, 2])
```

**3.2 Plotting the ROC Curve** You may refer to this article to learn about the ROC Curve

[25]:

```python
# Number of classes
n_classes = y_val.shape[1]

# Initialize dictionaries to store false positive rates (fpr), true positive␣
↪rates (tpr), and ROC AUC values
fpr = dict()
tpr = dict()
roc_auc = dict()

# Compute ROC curve and ROC area for each class
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_val[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_val.ravel(), y_score.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Aggregate all false positive rates
all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

# Initialize mean true positive rate (tpr)
mean_tpr = np.zeros_like(all_fpr)
for i in range(n_classes):
    mean_tpr += interp(all_fpr, fpr[i], tpr[i])

# Average it and compute macro-average ROC curve and ROC area
mean_tpr /= n_classes
fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

# Plotting the ROC curves
plt.rcParams['figure.figsize'] = [7, 5]  # Set figure size
lw = 2  # Line width
plt.figure()

# Plot micro-average ROC curve
plt.plot(fpr["micro"], tpr["micro"],
         label='micro-average (area = {})'
               ''.format(round(roc_auc["micro"], 5)),
         color='deeppink', linestyle=':', linewidth=4)

# Plot macro-average ROC curve
plt.plot(fpr["macro"], tpr["macro"],
         label='macro-average (area = {})'
               ''.format(round(roc_auc["macro"], 5)),
```

```
         color='navy', linestyle=':', linewidth=4)

# Plot ROC curves for each class
labels = ['axion', 'cdm', 'no']
colors = cycle(['aqua', 'darkorange', 'cornflowerblue'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=lw,
             label='{} (area = {})'
             ''.format(labels[i], round(roc_auc[i], 5)))

# Plot the diagonal line (chance level)
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc="lower right", prop={"size": 10})  # Add legend
plt.show()  # Display the plot
```



**3.3 Calculate evaluation metrics** You may refer to this article to learn more about these metrics

[26]:
```
# Convert the predicted probabilities to class labels
y_pred = np.argmax(y_score, axis=1)

# Convert y_val from one-hot encoding to class labels
y_true = np.argmax(y_val, axis=1)

# Calculate accuracy
accuracy = accuracy_score(y_true, y_pred)
print(f'Accuracy: {accuracy}')

# Calculate precision
precision = precision_score(y_true, y_pred, average='weighted')
print(f'Precision: {precision}')

# Calculate recall
recall = recall_score(y_true, y_pred, average='weighted')
print(f'Recall: {recall}')

# Calculate F1 score
f1 = f1_score(y_true, y_pred, average='weighted')
print(f'F1 Score: {f1}')
```

```
Accuracy: 0.9838666666666667
Precision: 0.9838511073349684
Recall: 0.9838666666666667
F1 Score: 0.9838559471663821
```

## 1.2 Submission Guidelines

- Fill out the pre- and post- hackathon surveys.
- You are required to submit a Google Colab Jupyter Notebook (.ipynb and pdf) clearly showing your implementation along with the evaluation metrics (ROC curve, AUC score, and other metrics) for the validation data.
- You must also submit the final trained model, including the model architecture and the trained weights ( For example: HDF5 file, .pb file, .pt file, etc. )
- You can use this example notebook as a template for your work.

  ***NOTE:*** You are free to use any ML framework such as PyTorch, Keras, TensorFlow, etc.

[27]:
```
torch.save(model, 'model.pt')
```