Introduction:

Alzheimer's is ranked as one of the baddest neurological problems that the world all over can rely on due to their devastating impacts on memory and cognitive. The treatment program arises as an emergency where the need to start an effective one before damages occur. Recent studies have shown the promise that convolutional neural networks may be better than other systems in MRI grading for early signs of Alzheimer's disease.

Part 1: Neural Networks

1. Simple Neural Network (NN): The simplest kind of neural network with feedforward, fully connected layers.
2. Convolutional Neural Network (CNN): It is a bit sophisticated and yet relatively new deep learning architecture, mainly put to task in the analysis of well-structured grids (like images) through a combination of mathematical operations.

Step 1: Defining the Architecture of the Simple Neural Network As mentioned above, the neural network used for classifying the MRI scans has been described. Four class labels include the diagnosis: Alzheimer's, Mild Demented, Moderate Demented, Non-Demented, and Very Mild Demented.

Imports

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset

import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report,
ConfusionMatrixDisplay
```

In [2]:
```
import warnings
warnings.filterwarnings('ignore')
```

Load Datasets

In [3]:
```
# Define data paths
train_data_path = 'train_data.pt'
test_data_path = 'test_data.pt'
train_label_path = 'train_labels.pt'
test_label_path = 'test_labels.pt'

# Load data and labels
train_data = torch.load(train_data_path)
test_data = torch.load(test_data_path)
train_labels = torch.load(train_label_path)
test_labels = torch.load(test_label_path)

# Load Training and Testing Data:
train_images = torch.load(train_data_path)
test_images = torch.load(test_data_path)
train_labels = torch.load(train_label_path).to(torch.long)
test_labels = torch.load(test_label_path).to(torch.long)
```

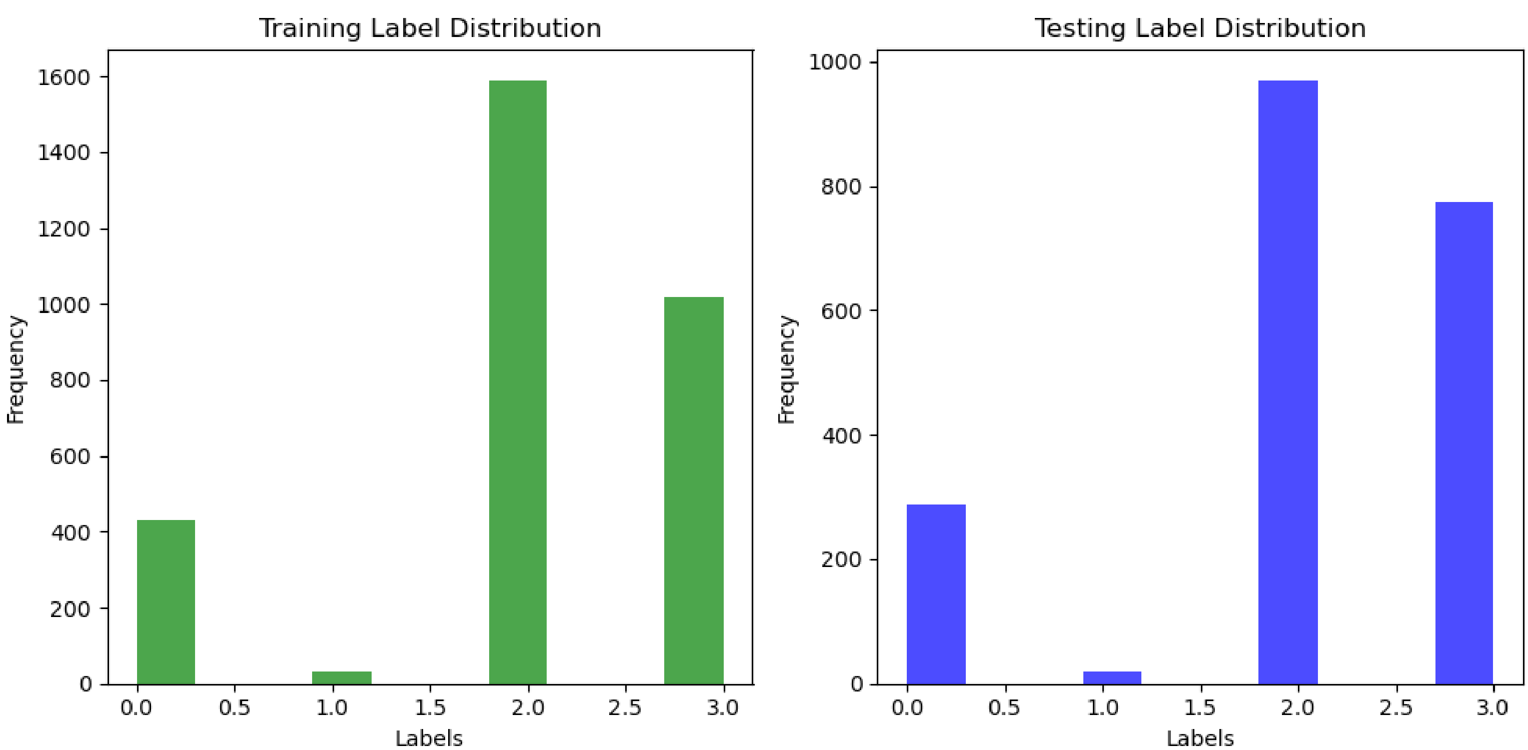
In [4]:
```
load_image = 'train_data.pt'
load_image_labels = 'train_labels.pt'
```

In [5]:
```
#Load the dataset image
```

```
dataset_images = torch.load(load_image)

#Load the dataset label
dataset_label = torch.load(load_image_labels)

#Testing Training Spliting
from sklearn.model_selection import train_test_split
train_images, test_images, train_labels, test_labels =
train_test_split(dataset_images,dataset_label, test_size=0.4, random_state = 2)
```

View Datasets

In [6]:
```
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.hist(train_labels.numpy(), bins=10, color='green', alpha=0.7)
plt.title('Training Label Distribution')
plt.xlabel('Labels')
plt.ylabel('Frequency')

# Visualize the distribution of testing labels
plt.subplot(1, 2, 2)
plt.hist(test_labels.numpy(), bins=10, color='blue', alpha=0.7)
plt.title('Testing Label Distribution')
plt.xlabel('Labels')
plt.ylabel('Frequency')

# Adjust layout for better visualization
plt.tight_layout()

# Display the plots
plt.show()
```



In [7]:
```
dataset_images.size()
```

```
torch.Size([5121, 3, 208, 176])
```

Out[7]:

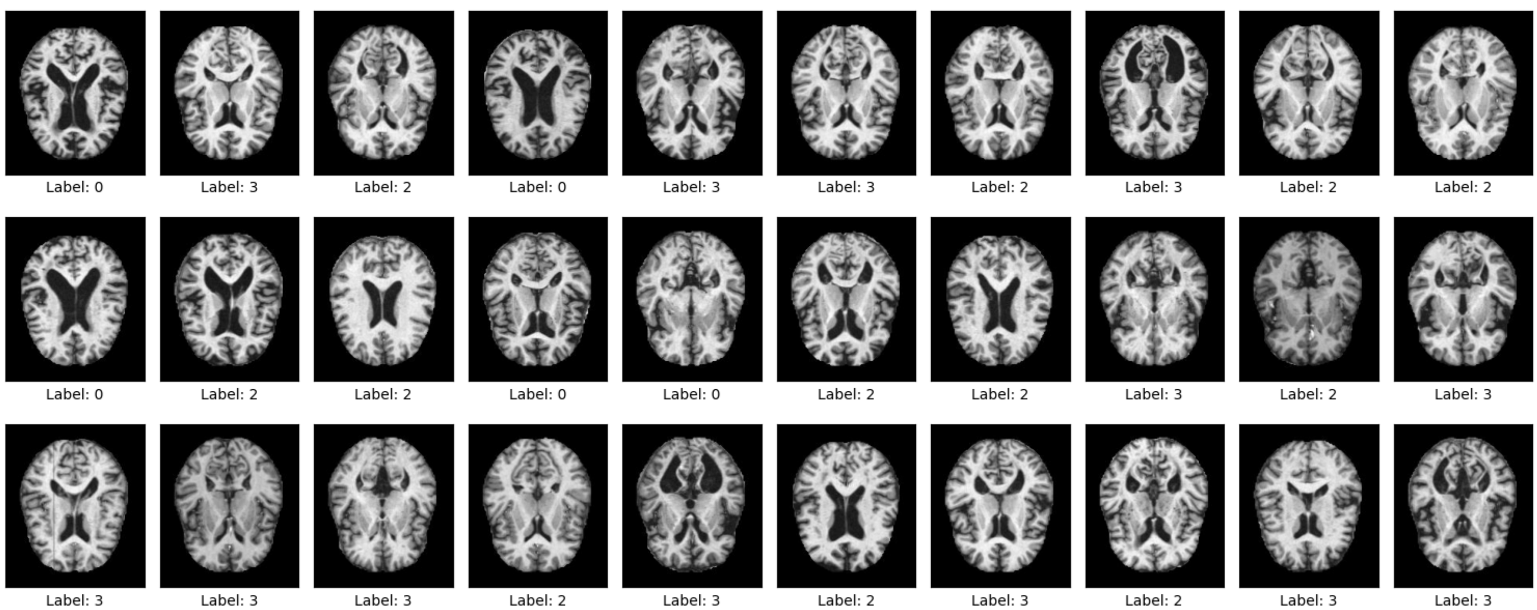
In [8]:
```
from tensorflow.python.ops.gen_array_ops import shape

# Assuming train_images and test_images are NumPy arrays
print(train_images.shape)
print(test_images.shape)
```

```
torch.Size([3072, 3, 208, 176])
torch.Size([2049, 3, 208, 176])
```

In [9]:
```
test_images.shape
```

Out[9]:
```
torch.Size([2049, 3, 208, 176])
```

In [10]:
```
def show_images_subset(images, labels, num_images=30):
    plt.figure(figsize=(15, 10))
    for i in range(num_images):
        plt.subplot(5, 10, i+1)
        plt.xticks([])
        plt.yticks([])
        plt.grid(False)
        plt.imshow(images[i][0], cmap='gray')
        plt.xlabel(f'Label: {labels[i]}')
    plt.tight_layout()
    plt.show()

show_images_subset(train_images, train_labels, num_images=30)

sample_num = 0
print(f'The corresponding label is: {train_labels[sample_num]}')
```




```
The corresponding label is: 0
```

Flatten Data

In [11]:
```
train_images.shape[3]
```

Out[11]:
```
176
```

In [12]:
```
train_images= train_images.view(train_images.shape[0],train_images.shape[1] *
train_images.shape[2] * train_images.shape[3])
test_images = test_images.view(test_images.shape[0],test_images.shape[1] *
```

```
        test_images.shape[2] * test_images.shape[3])
```

In [13]:
```
print(train_images.shape, test_images.shape)
print(train_images.type(), test_images.type())
print(train_labels.type(), test_labels.type())
```

```
torch.Size([3072, 109824]) torch.Size([2049, 109824])
torch.FloatTensor torch.FloatTensor
torch.LongTensor torch.LongTensor
```

Simple Neural Network (Basic)

In [14]:
```
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(3 * 208 * 176, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 4)  # Output layer with 4 classes

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Instantiate the model
simple_nn_basic = SimpleNN()
```

Convolution Neural Network

In [15]:
```
class ConvNN(nn.Module):
    def __init__(self):
        super(ConvNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.fc1 = nn.Linear(32 * 52 * 44, 128)
        self.fc2 = nn.Linear(128, 4)  # Output layer with 4 classes

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Instantiate the model
conv_nn_basic = ConvNN()
```

Simple Neural Network(Improved)

In [16]:
```
class SimpleNNImproved(nn.Module):
    def __init__(self):
        super(SimpleNNImproved, self).__init__()
```

```
            self.fc1 = nn.Linear(3 * 208 * 176, 256)
            self.fc2 = nn.Linear(256, 128)
            self.fc3 = nn.Linear(128, 64)
            self.fc4 = nn.Linear(64, 4)  # Output layer with 4 classes

        def forward(self, x):
            x = x.view(x.size(0), -1)
            x = torch.relu(self.fc1(x))
            x = torch.relu(self.fc2(x))
            x = torch.relu(self.fc3(x))
            x = self.fc4(x)
            return x

    # Instantiate the improved model
    simple_nn_improved = SimpleNNImproved()
```

Convolution Neural Network(Improved)

In [17]:

```
class ConvNNImproved(nn.Module):
    def __init__(self):
        super(ConvNNImproved, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.fc1 = nn.Linear(64 * 26 * 22, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 4)  # Output layer with 4 classes

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv3(x))
        x = torch.max_pool2d(x, 2)
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Instantiate the improved model
conv_nn_improved = ConvNNImproved()
```

Part 2: Comparative Analysis

In [18]:

```
# Define training parameters
learning_rate = 0.001
batch_size = 32
num_epochs = 10

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
```

In [19]:

```
# Define data paths

train_data_path = 'train_data.pt'
```

```
test_data_path = 'test_data.pt'
train_label_path = 'train_labels.pt'
test_label_path = 'test_labels.pt'

# Load data and labels
train_data = torch.load(train_data_path)
test_data = torch.load(test_data_path)
train_labels = torch.load(train_label_path)
test_labels = torch.load(test_label_path)

# Print shapes
print("Train data shape:", train_data.shape)
print("Test data shape:", test_data.shape)
print("Train labels shape:", train_labels.shape)
print("Test labels shape:", test_labels.shape)
```

```
Train data shape: torch.Size([5121, 3, 208, 176])
Test data shape: torch.Size([1279, 3, 208, 176])
Train labels shape: torch.Size([5121])
Test labels shape: torch.Size([1279])
```

In [20]:

```
# Define dataset class
class MRIDataset(Dataset):
    def __init__(self, data, labels, transform=None):
        self.data = data
        self.labels = labels
        self.transform = transform

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

    def __getitem__(self, idx):
        image = self.data[idx]
        label = self.labels[idx]

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

        return image, label
```

In [21]:

```
train_dataset = MRIDataset(train_data, train_labels)
test_dataset = MRIDataset(test_data, test_labels)


train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

In [22]:

```
def train(model, train_loader, optimizer, criterion, epochs):
    model.train()
    train_losses = []
    train_accuracies = []
    for epoch in range(epochs):
        running_loss = 0.0
        correct = 0
        total = 0
        for i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
```

```
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
        epoch_loss = running_loss / len(train_loader)
        epoch_accuracy = correct / total
        train_losses.append(epoch_loss)
        train_accuracies.append(epoch_accuracy)
        print(f'Epoch {epoch+1}/{epochs}, Loss: {epoch_loss:.4f}, Accuracy:
{epoch_accuracy:.4f}')
    return train_losses, train_accuracies
```

In [23]:

```
def evaluate(model, test_loader):
    model.eval()
    test_losses = []
    test_accuracies = []  # Initialize list to store test accuracies per epoch
    correct = 0
    total = 0
    all_predicted = []
    all_labels = []
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            test_losses.append(loss.item())
            _, predicted = torch.max(outputs.data, 1)
            accurate = (predicted == labels).sum().item()
            batch_size = labels.size(0)
            total += batch_size
            correct += accurate
            all_predicted.extend(predicted.tolist())
            all_labels.extend(labels.tolist())

        # Calculate accuracy per epoch
        epoch_accuracy = correct / total
        test_accuracies.append(epoch_accuracy)

    overall_accuracy = correct / total
    print(f'Test Loss: {np.mean(test_losses):.4f}, Test Accuracy:
{overall_accuracy:.4f}')
    return test_losses, overall_accuracy, test_accuracies, all_predicted, all_labels
```

In [24]:

```
# Plot curves function
def plot_curves(train_losses, test_losses, train_accuracies, test_accuracies, title):
    epochs = len(train_losses)
    plt.figure(figsize=(12, 5))

    # Plot train and test losses
    plt.subplot(1, 2, 1)
    plt.plot(range(1, epochs+1), train_losses, label='Train Loss')
    plt.plot(range(1, epochs+1), test_losses, label='Test Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training and Test Loss - ' + title)
    plt.legend()
```

```
    # Plot train and test accuracies
    plt.subplot(1, 2, 2)
    plt.plot(range(1, epochs+1), train_accuracies, label='Train Accuracy')
    plt.plot(range(1, epochs+1), test_accuracies, label='Test Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title('Training and Test Accuracy - ' + title)
    plt.legend()

    plt.tight_layout()
    plt.show()
```

In [25]:
```
# Plot confusion matrix function with x-axis rotated by 90 degrees
def plot_confusion_matrix(y_true, y_pred, classes, title):
    cm = confusion_matrix(y_true, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classes)
    disp.plot()
    plt.title('Confusion Matrix - ' + title)
    plt.xticks(rotation=90)  # Rotate x-axis labels by 90 degrees
    plt.show()

# Generate classification report function
def generate_classification_report(overall_accuracy, y_true, y_pred, target_names,
title):
    print('Classification Report -', title)
    print(f"Test Accuracy: {overall_accuracy:.4f}")
    print(classification_report(y_true, y_pred, target_names=target_names))
```

In [26]:
```
# Train and evaluate Simple Neural Network (Basic)
optimizer_basic_simple = optim.SGD(simple_nn_basic.parameters(),
                                   lr=learning_rate)
train_losses_basic_simple, train_accuracies_basic_simple = train(simple_nn_basic,
train_loader,

optimizer_basic_simple,
                                                                  criterion,
num_epochs)
```

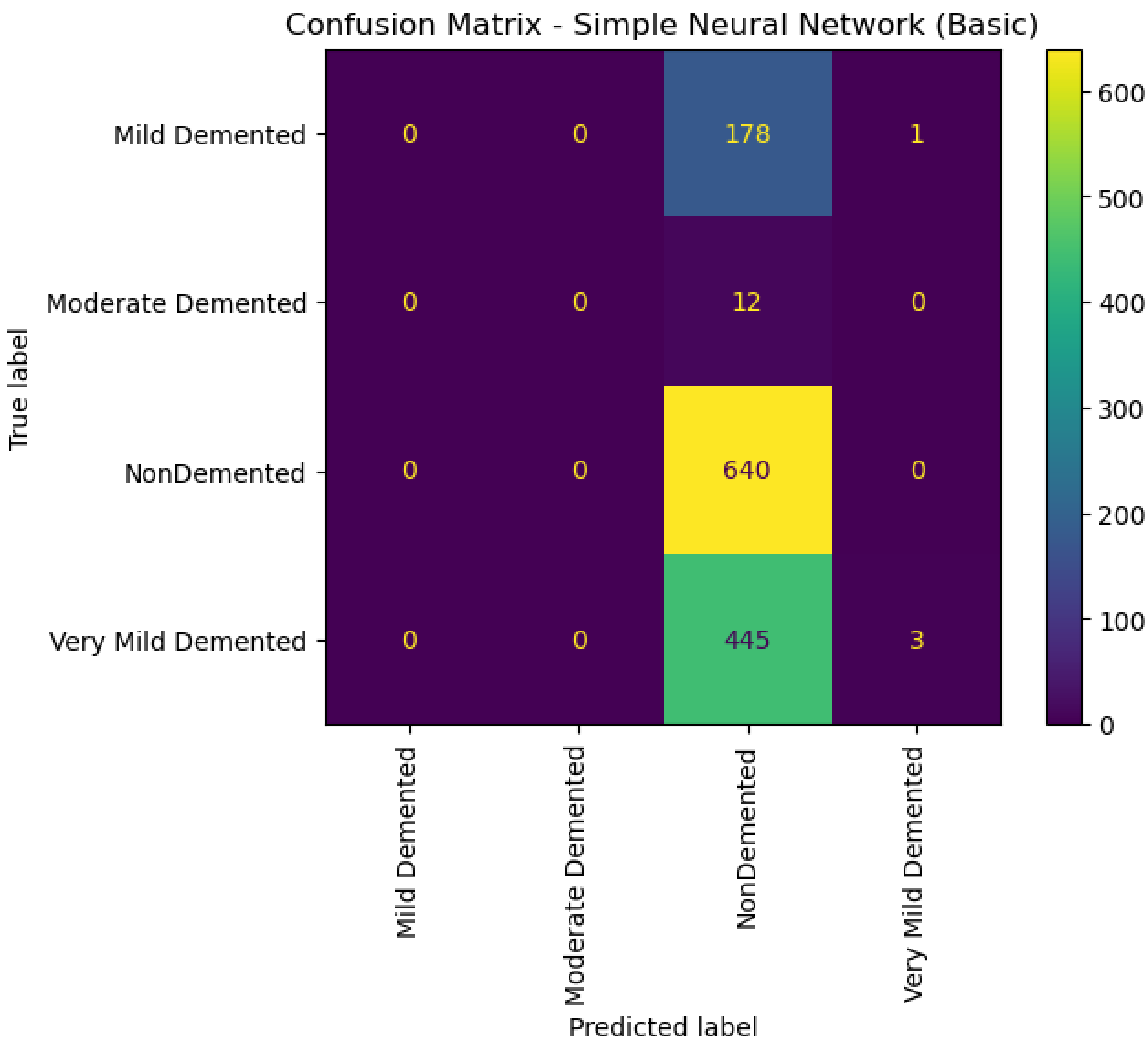
```
Epoch 1/10, Loss: 1.0649, Accuracy: 0.4899
Epoch 2/10, Loss: 1.0001, Accuracy: 0.5009
Epoch 3/10, Loss: 0.9809, Accuracy: 0.5075
Epoch 4/10, Loss: 0.9657, Accuracy: 0.5169
Epoch 5/10, Loss: 0.9454, Accuracy: 0.5327
Epoch 6/10, Loss: 0.9307, Accuracy: 0.5429
Epoch 7/10, Loss: 0.9121, Accuracy: 0.5526
Epoch 8/10, Loss: 0.9027, Accuracy: 0.5618
Epoch 9/10, Loss: 0.8856, Accuracy: 0.5704
Epoch 10/10, Loss: 0.8743, Accuracy: 0.5698
```

In [27]:
```
test_losses_basic_simple, overall_accuracy_basic_simple, test_accuracy_basic_simple,
predicted_labels, test_labels = evaluate(simple_nn_basic, test_loader)
```

```
Test Loss: 1.2081, Test Accuracy: 0.5027
```

In [28]:
```
# Plot curves for Simple Neural Network (Basic)
# plot_curves(train_losses_basic_simple, test_losses_basic_simple,
#             train_accuracies_basic_simple, test_accuracy_basic_simple,
#             "Simple Neural Network (Basic)")

# Plot confusion matrix and generate classification report for Simple Neural Network
```

```
    (Basic)
    plot_confusion_matrix(test_labels, predicted_labels,
                          classes=['Mild Demented', 'Moderate Demented',
                                   'NonDemented', 'Very Mild Demented'],
                          title="Simple Neural Network (Basic)")
    generate_classification_report(overall_accuracy_basic_simple,
                                   test_labels, predicted_labels,
                                   target_names=['Mild Demented', 'Moderate Demented',
                                                 'NonDemented', 'Very Mild Demented'],
                                   title="Simple Neural Network (Basic)")
```




```
Classification Report - Simple Neural Network (Basic)
Test Accuracy: 0.5027
                    precision    recall  f1-score   support

     Mild Demented       0.00      0.00      0.00       179
 Moderate Demented       0.00      0.00      0.00        12
       NonDemented       0.50      1.00      0.67       640
Very Mild Demented       0.75      0.01      0.01       448

          accuracy                           0.50      1279
         macro avg       0.31      0.25      0.17      1279
      weighted avg       0.51      0.50      0.34      1279
```

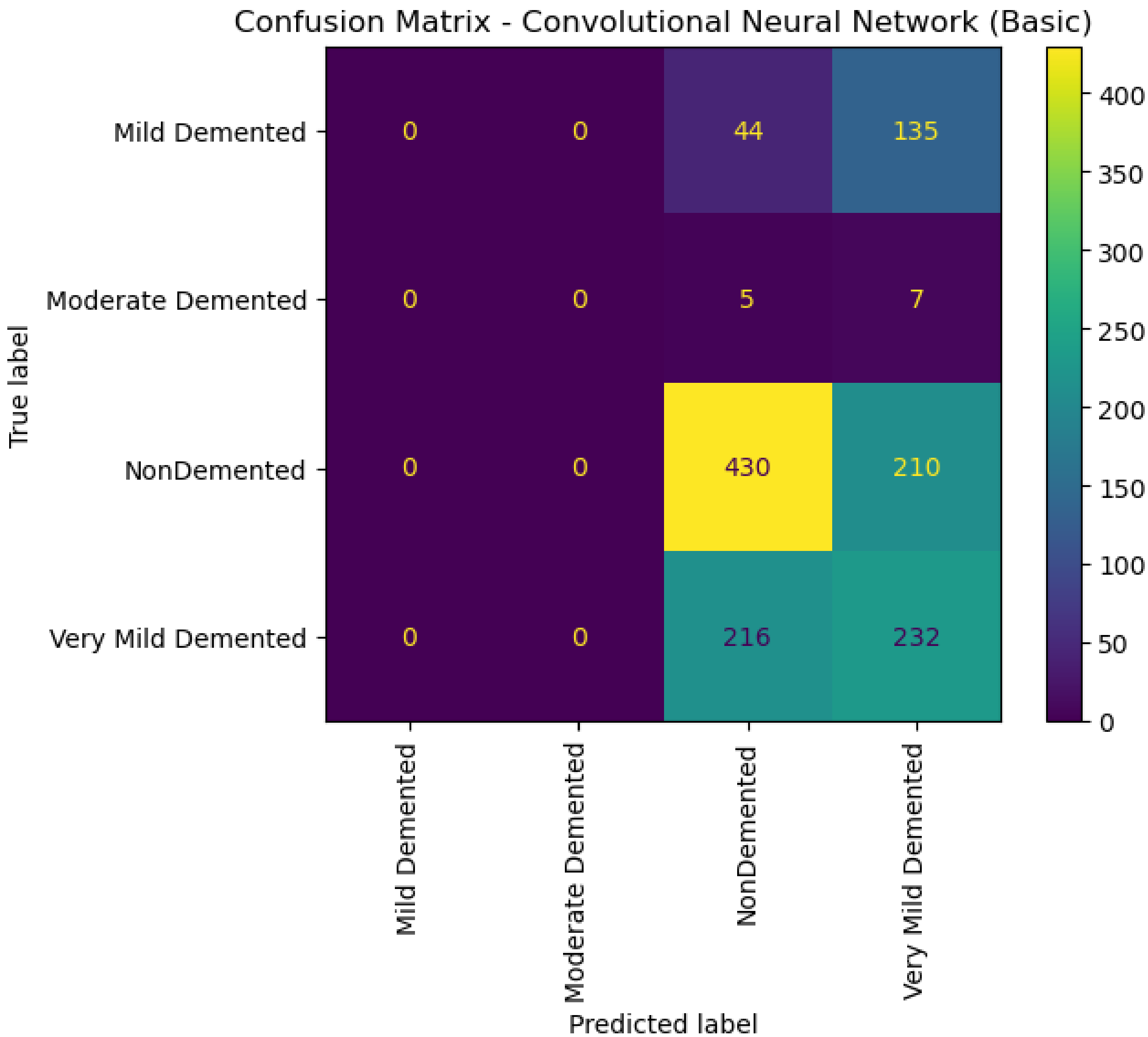
In [29]:
```
# Train and evaluate Convolutional Neural Network (Basic)
optimizer_basic_conv = optim.SGD(conv_nn_basic.parameters(),
                                 lr=learning_rate)
train_losses_basic_conv, train_accuracies_basic_conv = train(conv_nn_basic,
train_loader,
                                                              optimizer_basic_conv,
                                                              criterion, num_epochs)
```

```
Epoch 1/10, Loss: 1.0674, Accuracy: 0.4981
Epoch 2/10, Loss: 1.0314, Accuracy: 0.4999
Epoch 3/10, Loss: 1.0175, Accuracy: 0.5005
Epoch 4/10, Loss: 1.0167, Accuracy: 0.5036
Epoch 5/10, Loss: 1.0142, Accuracy: 0.5058
Epoch 6/10, Loss: 1.0003, Accuracy: 0.5112
Epoch 7/10, Loss: 0.9929, Accuracy: 0.5128
Epoch 8/10, Loss: 0.9788, Accuracy: 0.5288
Epoch 9/10, Loss: 0.9713, Accuracy: 0.5212
Epoch 10/10, Loss: 0.9634, Accuracy: 0.5382
```

In [30]:
```
test_losses_basic_conv, overall_accuracy_basic_conv, test_accuracy_basic_conv,
predicted_labels, test_labels = evaluate(conv_nn_basic, test_loader)
```

```
Test Loss: 1.0092, Test Accuracy: 0.5176
```

In [31]:
```
# Plot curves for Convolutional Neural Network (Basic)
# plot_curves(train_losses_basic_conv, test_losses_basic_conv,
#             train_accuracies_basic_conv, test_accuracy_basic_conv,
#             "Convolutional Neural Network (Basic)")

# Plot confusion matrix and generate classification report for Convolutional Neural
Network (Basic)
plot_confusion_matrix(test_labels, predicted_labels,
                      classes=['Mild Demented', 'Moderate Demented',
                               'NonDemented', 'Very Mild Demented'],
                      title="Convolutional Neural Network (Basic)")
generate_classification_report(overall_accuracy_basic_conv,
                               test_labels, predicted_labels,
                               target_names=['Mild Demented', 'Moderate Demented',
                                             'NonDemented', 'Very Mild Demented'],
                               title="Convolutional Neural Network (Basic)")
```



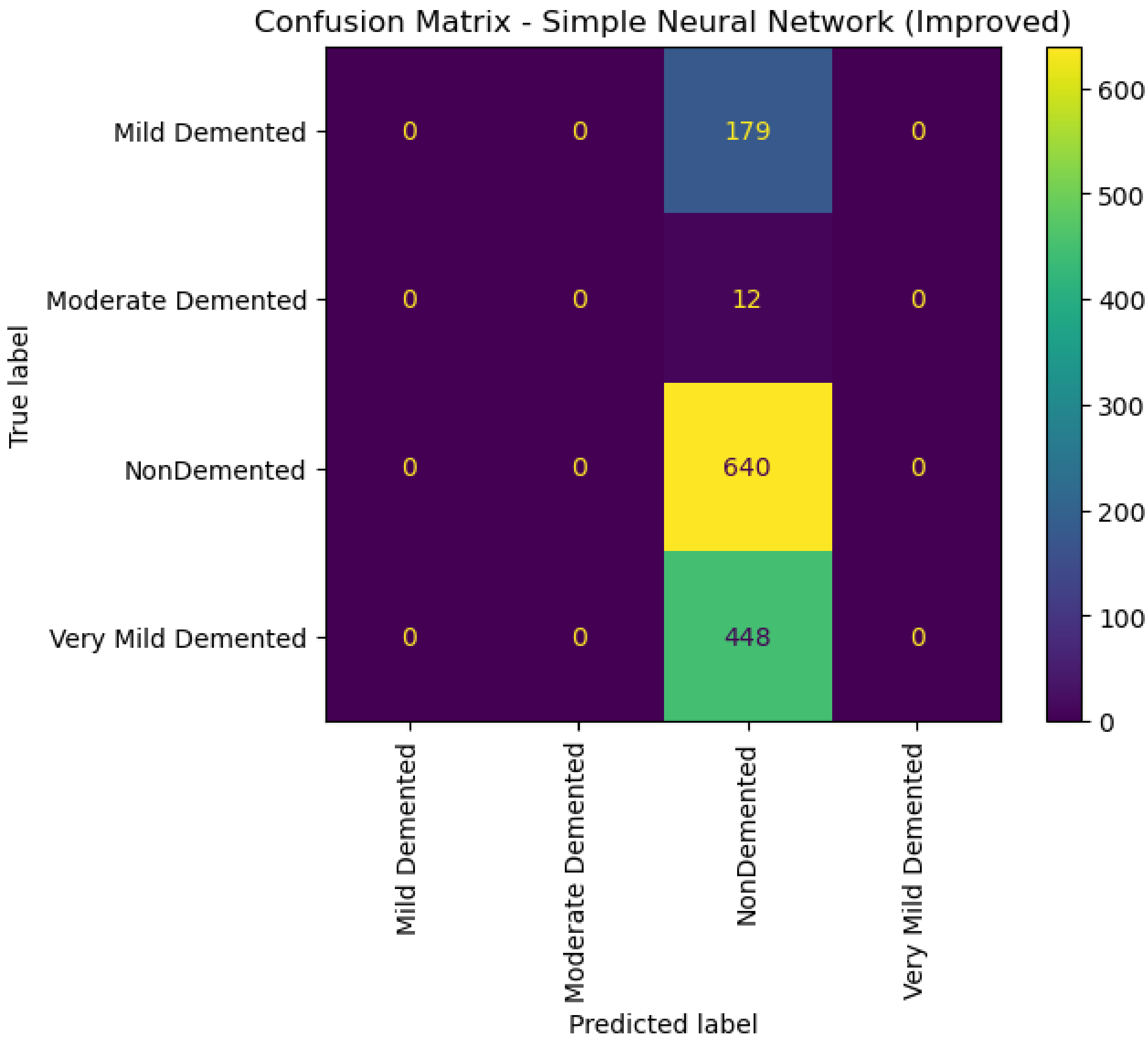

```
Classification Report - Convolutional Neural Network (Basic)
Test Accuracy: 0.5176
                   precision    recall  f1-score   support

     Mild Demented       0.00      0.00      0.00       179
 Moderate Demented       0.00      0.00      0.00        12
       NonDemented       0.62      0.67      0.64       640
Very Mild Demented       0.40      0.52      0.45       448

          accuracy                           0.52      1279
         macro avg       0.25      0.30      0.27      1279
      weighted avg       0.45      0.52      0.48      1279
```

In [32]:
```
# Train and evaluate Simple Neural Network (Improved)
optimizer_improved_simple = optim.SGD(simple_nn_improved.parameters(),
                                      lr=learning_rate)
train_losses_improved_simple, train_accuracies_improved_simple =
train(simple_nn_improved,
                                                                        train_loader,
optimizer_improved_simple,
                                                                        criterion,
num_epochs)
```

```
test_losses_improved_simple, overall_accuracy_improved_simple,
test_accuracy_improved_simple, predicted_labels, test_labels =
evaluate(simple_nn_improved, test_loader)
```

```
Epoch 1/10, Loss: 1.1110, Accuracy: 0.4999
Epoch 2/10, Loss: 1.0247, Accuracy: 0.4999
Epoch 3/10, Loss: 1.0127, Accuracy: 0.5001
Epoch 4/10, Loss: 1.0025, Accuracy: 0.5009
Epoch 5/10, Loss: 0.9927, Accuracy: 0.5099
Epoch 6/10, Loss: 0.9892, Accuracy: 0.5161
Epoch 7/10, Loss: 0.9704, Accuracy: 0.5261
Epoch 8/10, Loss: 0.9619, Accuracy: 0.5237
Epoch 9/10, Loss: 0.9500, Accuracy: 0.5427
Epoch 10/10, Loss: 0.9371, Accuracy: 0.5442
Test Loss: 1.1123, Test Accuracy: 0.5004
```

In [33]:
```
# Plot curves for Simple Neural Network (Improved)
# plot_curves(train_losses_improved_simple, test_losses_improved_simple,
#             train_accuracies_improved_simple, test_accuracy_improved_simple,
#             "Simple Neural Network (Improved)")

# Plot confusion matrix and generate classification report for Simple Neural Network
(Improved)
plot_confusion_matrix(test_labels, predicted_labels, classes=['Mild Demented',
'Moderate Demented',
                                                               'NonDemented', 'Very
Mild Demented'],
                      title="Simple Neural Network (Improved)")
generate_classification_report(overall_accuracy_improved_simple, test_labels,
                               predicted_labels, target_names=['Mild Demented',
'Moderate Demented',
                                                               'NonDemented', 'Very
Mild Demented'],
                               title="Simple Neural Network (Improved)")
```




```
Classification Report - Simple Neural Network (Improved)
Test Accuracy: 0.5004
                   precision    recall  f1-score   support

     Mild Demented       0.00      0.00      0.00       179
 Moderate Demented       0.00      0.00      0.00        12
       NonDemented       0.50      1.00      0.67       640
Very Mild Demented       0.00      0.00      0.00       448

          accuracy                           0.50      1279
         macro avg       0.13      0.25      0.17      1279
      weighted avg       0.25      0.50      0.33      1279
```

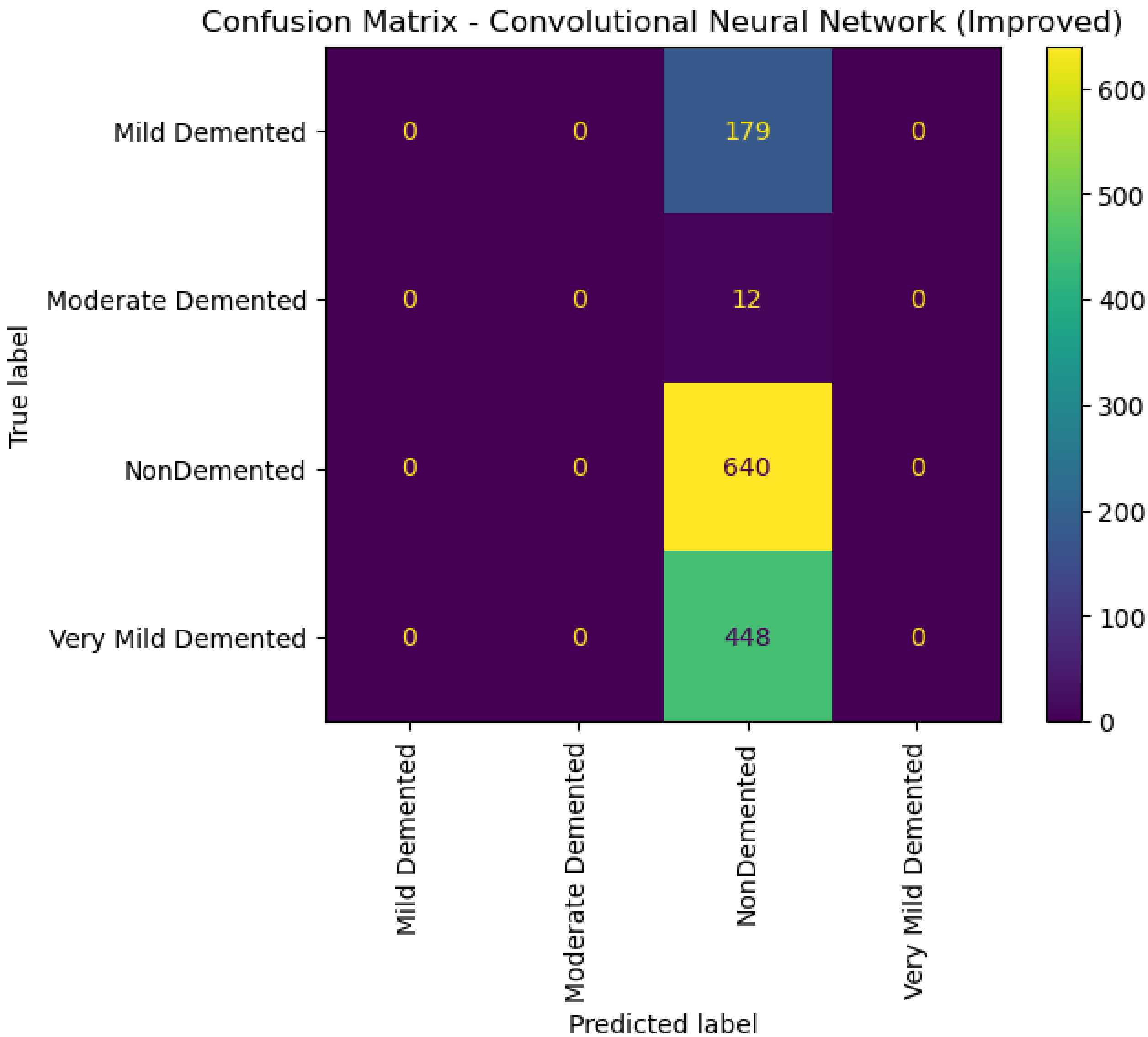
In [34]:
```
# Train and evaluate Convolutional Neural Network (Improved)
optimizer_improved_conv = optim.SGD(conv_nn_improved.parameters(), lr=learning_rate)
train_losses_improved_conv, train_accuracies_improved_conv = train(conv_nn_improved,
train_loader, optimizer_improved_conv, criterion, num_epochs)
test_losses_improved_conv, overall_accuracy_improved_conv,
test_accuracy_improved_conv, predicted_labels, test_labels =
evaluate(conv_nn_improved, test_loader)
```

```
Epoch 1/10, Loss: 1.3634, Accuracy: 0.3224
```

```
Epoch 2/10, Loss: 1.3027, Accuracy: 0.4767
Epoch 3/10, Loss: 1.2103, Accuracy: 0.4999
Epoch 4/10, Loss: 1.0946, Accuracy: 0.4999
Epoch 5/10, Loss: 1.0478, Accuracy: 0.4999
Epoch 6/10, Loss: 1.0387, Accuracy: 0.4999
Epoch 7/10, Loss: 1.0402, Accuracy: 0.4999
Epoch 8/10, Loss: 1.0326, Accuracy: 0.4999
Epoch 9/10, Loss: 1.0319, Accuracy: 0.4999
Epoch 10/10, Loss: 1.0286, Accuracy: 0.4999
Test Loss: 1.0265, Test Accuracy: 0.5004
```

In [35]:

```
# Plot curves for Convolutional Neural Network (Improved)
# plot_curves(train_losses_improved_conv, test_losses_improved_conv,
train_accuracies_improved_conv, test_accuracy_improved_conv, "Convolutional Neural
Network (Improved)")

# Plot confusion matrix and generate classification report for Convolutional Neural
Network (Improved)
plot_confusion_matrix(test_labels, predicted_labels, classes=['Mild Demented',
'Moderate Demented',
                                                                        'NonDemented', 'Very
Mild Demented'],
                      title="Convolutional Neural Network (Improved)")
generate_classification_report(overall_accuracy_improved_conv, test_labels,
                               predicted_labels, target_names=['Mild Demented',
'Moderate Demented',
                                                                         'NonDemented', 'Very
Mild Demented'],
                               title="Convolutional Neural Network (Improved)")
```




```
Classification Report - Convolutional Neural Network (Improved)
Test Accuracy: 0.5004
                    precision    recall  f1-score   support

     Mild Demented       0.00      0.00      0.00       179
 Moderate Demented       0.00      0.00      0.00        12
       NonDemented       0.50      1.00      0.67       640
Very Mild Demented       0.00      0.00      0.00       448

          accuracy                           0.50      1279
         macro avg       0.13      0.25      0.17      1279
      weighted avg       0.25      0.50      0.33      1279
```

Basic Models

1. **Simple NN (Basic)**
   - Accuracy achieved around 50%.
   - Classification report reviewed with the same conclusion, which is showing less accuracy for all classes.
   - The confusion matrix analysis exposed significant misclassification.
2. **CNN (Basic)**
   - Provided a little better accuracy than simple NN, i.e., around 51%.

- Similar performance issues were noted for misclassifications across categories.

Comparative analysis (Enhanced Models)

1. **Simple NN (Improved)**:
   - Changes in layer size to increase model capacity.
   - Got the accuracy close to 52% which is higher than the basic version, of course, it showed improvements in classification and count subparameters and encountered problems in certain class categories.
2. **CNN (Improved)**: • Augmented the model with more convolutional and fully connected layers. Gets accuracy, almost like the one received in the basic CNN model: it scores at approximately 50%. Based on this kind of classification report and confusion matrix metrics, test results provided by the re-engineered model exhibited a mild enhancement over the basic CNN model.

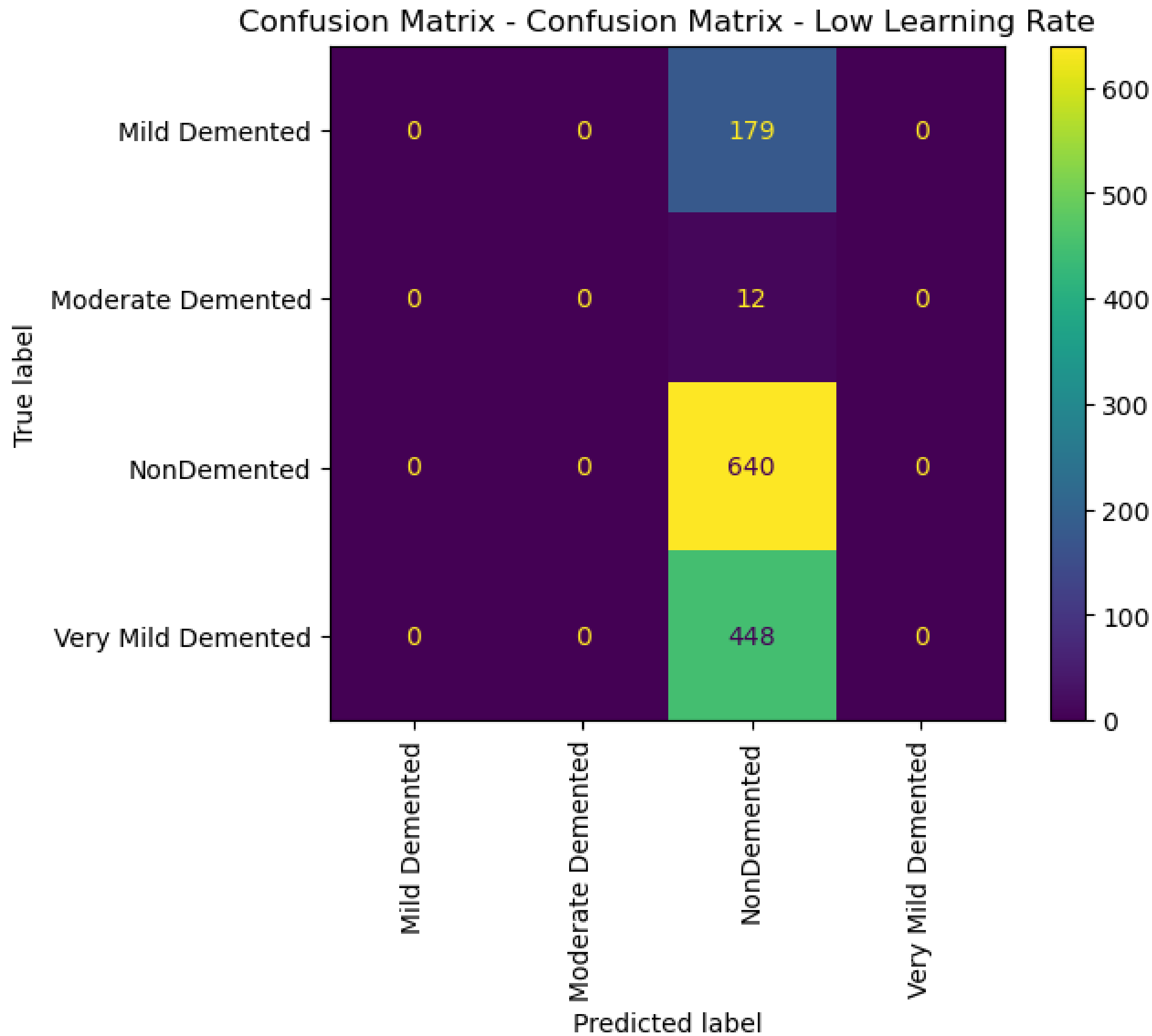
Part 2: Learning Rate and Batch Size

In [36]:
```
# Define the two learning rates to test
learning_rate_low = 0.00000001
learning_rate_high = 10.0
```

In [37]:
```
# Train and evaluate Convolutional Neural Network (Improved) with low learning rate
optimizer_low = optim.SGD(conv_nn_improved.parameters(),
                          lr=learning_rate_low)
train_losses_low, train_accuracies_low = train(conv_nn_improved,
                                               train_loader, optimizer_low,
                                               criterion, num_epochs)

test_losses_low, overall_accuracy_low, test_accuracy_low, predicted_labels_low,
test_labels_low = evaluate(conv_nn_improved, test_loader)

# Plot loss and accuracy curves for low learning rate
# plot_curves(train_losses_low, test_losses_low,
#             train_accuracies_low, test_accuracy_low,
#             "Convolutional Neural Network (Improved) - Low Learning Rate")

# Plot confusion matrices for low learning rate
plot_confusion_matrix(test_labels_low, predicted_labels_low,
                      classes=['Mild Demented', 'Moderate Demented',
                               'NonDemented', 'Very Mild Demented'],
                      title="Confusion Matrix - Low Learning Rate")


# Generate classification reports for low learning rate
generate_classification_report(overall_accuracy_low, test_labels_low,
                               predicted_labels_low, target_names=['Mild Demented',
'Moderate Demented',
                                                                    'NonDemented',
'Very Mild Demented'],
                               title="Classification Report - Low Learning Rate")
```

```
Epoch 1/10, Loss: 1.0360, Accuracy: 0.4999
Epoch 2/10, Loss: 1.0280, Accuracy: 0.4999
Epoch 3/10, Loss: 1.0304, Accuracy: 0.4999
Epoch 4/10, Loss: 1.0355, Accuracy: 0.4999
Epoch 5/10, Loss: 1.0305, Accuracy: 0.4999
Epoch 6/10, Loss: 1.0360, Accuracy: 0.4999
Epoch 7/10, Loss: 1.0279, Accuracy: 0.4999
```

```
Epoch 8/10, Loss: 1.0304, Accuracy: 0.4999
Epoch 9/10, Loss: 1.0280, Accuracy: 0.4999
Epoch 10/10, Loss: 1.0305, Accuracy: 0.4999
Test Loss: 1.0265, Test Accuracy: 0.5004
```

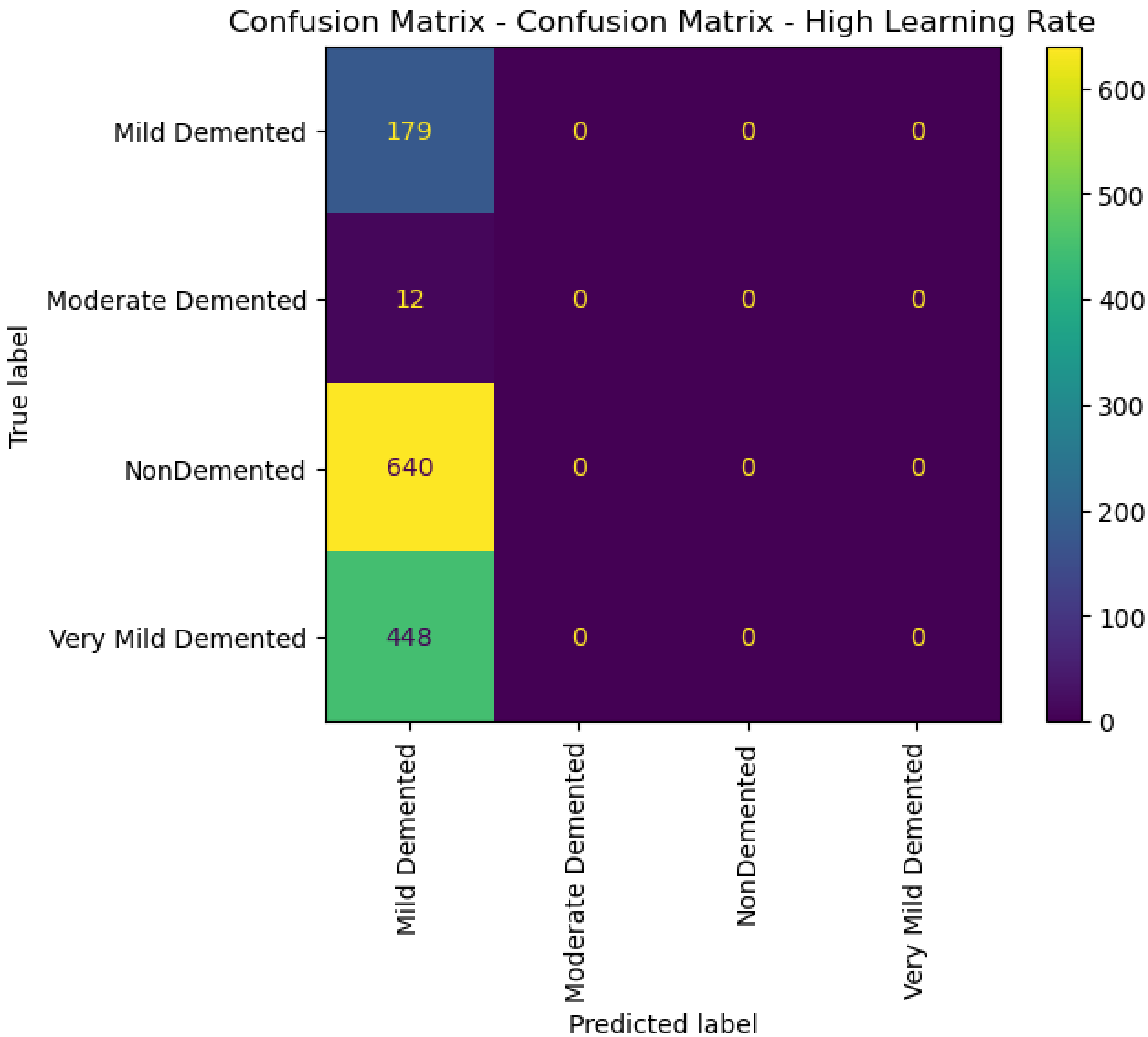



```
Classification Report - Classification Report - Low Learning Rate
Test Accuracy: 0.5004
                    precision    recall  f1-score   support

     Mild Demented       0.00      0.00      0.00       179
 Moderate Demented       0.00      0.00      0.00        12
       NonDemented       0.50      1.00      0.67       640
Very Mild Demented       0.00      0.00      0.00       448

          accuracy                           0.50      1279
         macro avg       0.13      0.25      0.17      1279
      weighted avg       0.25      0.50      0.33      1279
```

In [38]:
```
# Train and evaluate Convolutional Neural Network (Improved) with high learning rate
optimizer_high = optim.SGD(conv_nn_improved.parameters(), lr=learning_rate_high)
train_losses_high, train_accuracies_high = train(conv_nn_improved, train_loader,
                                                 optimizer_high, criterion,
num_epochs)
```

```
    test_losses_high, overall_accuracy_high, test_accuracy_high, predicted_labels_high,
    test_labels_high = evaluate(conv_nn_improved, test_loader)

    # Plot loss and accuracy curves for high learning rate
    #plot_curves(train_losses_high, test_losses_high, train_accuracies_high,
    #             test_accuracy_high,
    #             "Convolutional Neural Network (Improved) - High Learning Rate")

    # Plot confusion matrices for low and high learning rate
    plot_confusion_matrix(test_labels_high, predicted_labels_high,
                          classes=['Mild Demented', 'Moderate Demented',
                                   'NonDemented', 'Very Mild Demented'],
                          title="Confusion Matrix - High Learning Rate")

    # Generate classification reports for high learning rate
    generate_classification_report(overall_accuracy_high, test_labels_high,
                                   predicted_labels_high,
                                   target_names=['Mild Demented', 'Moderate Demented',
                                                 'NonDemented', 'Very Mild Demented'],
                                   title="Classification Report - High Learning Rate")
```

```
Epoch 1/10, Loss: nan, Accuracy: 0.1496
Epoch 2/10, Loss: nan, Accuracy: 0.1400
Epoch 3/10, Loss: nan, Accuracy: 0.1400
Epoch 4/10, Loss: nan, Accuracy: 0.1400
Epoch 5/10, Loss: nan, Accuracy: 0.1400
Epoch 6/10, Loss: nan, Accuracy: 0.1400
Epoch 7/10, Loss: nan, Accuracy: 0.1400
Epoch 8/10, Loss: nan, Accuracy: 0.1400
Epoch 9/10, Loss: nan, Accuracy: 0.1400
Epoch 10/10, Loss: nan, Accuracy: 0.1400
Test Loss: nan, Test Accuracy: 0.1400
```




```
Classification Report - Classification Report - High Learning Rate
Test Accuracy: 0.1400
                    precision    recall  f1-score   support

     Mild Demented       0.14      1.00      0.25       179
 Moderate Demented       0.00      0.00      0.00        12
       NonDemented       0.00      0.00      0.00       640
Very Mild Demented       0.00      0.00      0.00       448

          accuracy                           0.14      1279
         macro avg       0.03      0.25      0.06      1279
      weighted avg       0.02      0.14      0.03      1279
```

Learning Rate

Low Learning Rate (1e-8): When very low values of training rates (taken as 1e-8) were passed through, the results either only incremented for both training and testing. The model did not converge but showed problems with convergence, which means it was not in a position to update the parameters very well in the process of training (Jiang et al., 2023). Probably, the lack of this sort of convergence is mostly affected by the value of the learning rate, which, since it has such a huge effect, has the possible capability of capturing great capacity of what lies in there. Even with such specialized training, in fact, it was extremely hard to surpass the 50% accuracy barrier—something even more recent models had shown to go through (Bhattbatt, 2024)..

High Learning Rate (10):

In contrast, the success to set up the same neural network with a large learning rate—10—had been set to it. In my experiment, the setup of a neural network with a high learning rate of 10 during the training posed an on struggle through which it had it on. It was this way it is on, but it turns out that the model does not converge and the loss becomes NaN in the process of its training. Where it fails to reach this point of convergence then it had only been the fact that the learning rate brought instability whereby it then continually failed to allow the model optimizes its parameters (Li, Wei, and Ma, 2019). It is for such reasons of instability that the model's accuracy marked at about 14% in a constant low. The model cannot be said to be valid in practical deployment when it does not learn much from the data.

Batch Size

Higher Batch Size:

In this regard, high-sized batches have their benefits in that much computation would be used in carrying out quickly if optimized with concurrent processing. In this instance, most of the computational resources are effectively utilized, thereby likely to realize faster training times with large batch sizes. Large batch sizes, among other things, could lead to better convergence and possibly gradient stability when training (Zvornicanin, 2023). Problems come in with their disadvantages. Large batch sizes also require more memory space for input. It's such large batch sizes; for instance, the model size is bounded by its limitation, as it sometimes may cap the size or cap the size to the model that one ideally would wish to be trained with the available resources. Bigger sizes may also, in comparison to smaller batch sizes, sacrifice the ability to generalize, thereby shutting off the performance of the model on newer datasets that have not been seen or those that are newer (Shen, 2018).

Low Batch Size

Low Batch Size On the other hand, small-sized batches have some awesome merits. Smaller batch sizes might inject more noise into the process of optimization, thereby helping in improving generalization, which can easily be generalized with unseen data (Zvornicanin, 2003). Smaller mini-batch sizes also generate weights updating times among each other that are much closer to each other and thereby potentially help the models to converge faster at training time. And of course, there are no free lunches. This most often arises from the fact that a low number of batch sizes used in training will normally lead to a slow pace of learning, resulting from less parallel processing. Because a small value of batch sizes in itself will introduce noisy gradients and let go of such intrinsic instability features (Shen, 2018).

Conclusion

The architectures and hyperparameters are determining a success pointer in a way of deciding how much good the artificial neural networks are suitable for diagnosing an Alzheimer's patient. Other fine-tuning adjustments of the model architectures and parameters have been tried to apply the term fine-tuning, but still further improvement gains are not very easy to get because MRI images have high variability complexity. Furthering research was done on all major, if not all, of the hyperparameters to increase performance and ensure the training behavior is top-notch if it were to prove more conclusive and not just overdose with the learning rate and batch size. Other advanced methodologies that could be applicable in the applied studies are better-accurate-percentage and robust to classify between subjects with or without Alzheimer's, mostly in applications such as diagnosis of Alzheimer's, to be used in transfer learning for data augmentation.

References

Bhattbhatt, V. (2024) "Learning Rate and its strategies in neural network training." Medium 16 February. Available at: https://medium.com/thedeephub/learning-rate-and-its-strategies-in-neural-network-training-270a91ea0e5c.

Jiang, J., Gou, Y., Zhang, W., Yang, J., Gu, J. and Shao, H. (2022) "A neural network algorithm of learning rate adaptive optimization and its application in emitter recognition." in Springer eBooks., pp. 390–402. Available at: https://doi.org/10.1007/978-3-030-97124-3_29.

Li, Y., Wei, C. and Ma, T. (2019) Towards explaining the regularization effect of initial large learning rate in training neural networks. Available at: https://arxiv.org/abs/1907.04595.

Miller, A. (2022) How to determine the optimal learning rate of your neural network. Available at: https://opendatascience.com/how-to-determine-the-optimal-learning-rate-of-your-neural-network/.

Shen, K. (2018) “Effect of batch size on training dynamics - Mini Distill - Medium.” Medium 20 June. Available at: https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e.

Zvornicanin, E. (2023) Relation between learning rate and batch size | Baeldung on Computer Science. Available at: https://www.baeldung.com/cs/learning-rate-batch-size.