Part 1: CNN Classifier

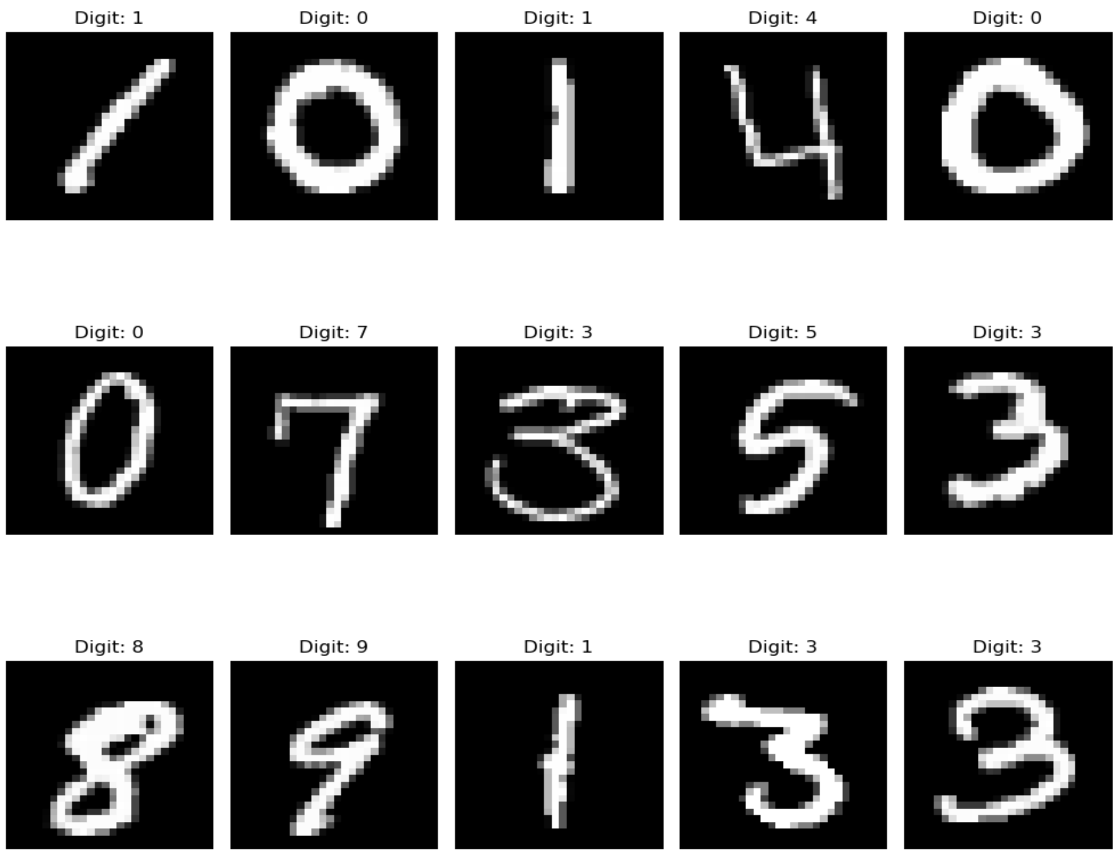
Comparative Analysis of CNN and Faster R-CNN Models for Image Classification and Object Detection

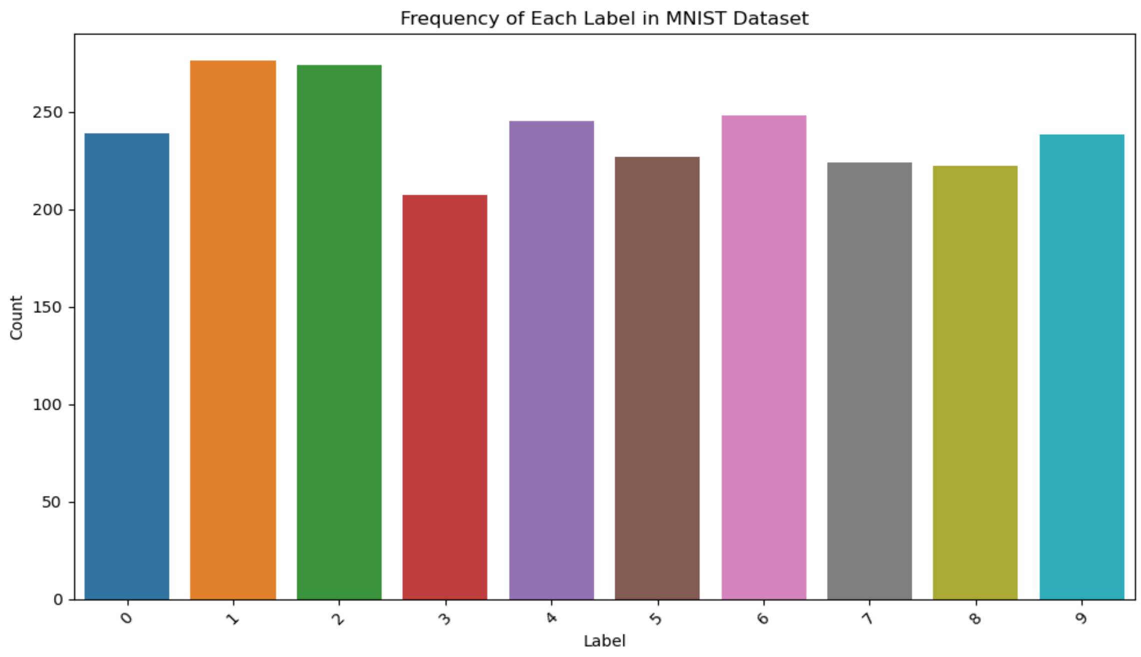
Objective:

The main objective of this study is to explore the capabilities of the PyTorch library for building neural architectures such as Convolutional Neural Networks (CNN) and Region-based Convolutional Neural Networks (Faster R-CNN) for computer vision tasks. Specifically, the study aims to:

1. Implement a CNN architecture for classifying the MNIST dataset.
2. Implement a Faster R-CNN architecture for object detection.
3. Compare the performance of the CNN and Faster R-CNN models using various metrics such as accuracy, F1 score, loss, and training time.
4. Fine-tune pre-trained models (VGG16 and AlexNet) on the MNIST dataset and compare the results with CNN and Faster R-CNN models.

Dataset: MNIST Dataset (Digit Recognizer | Kaggle)





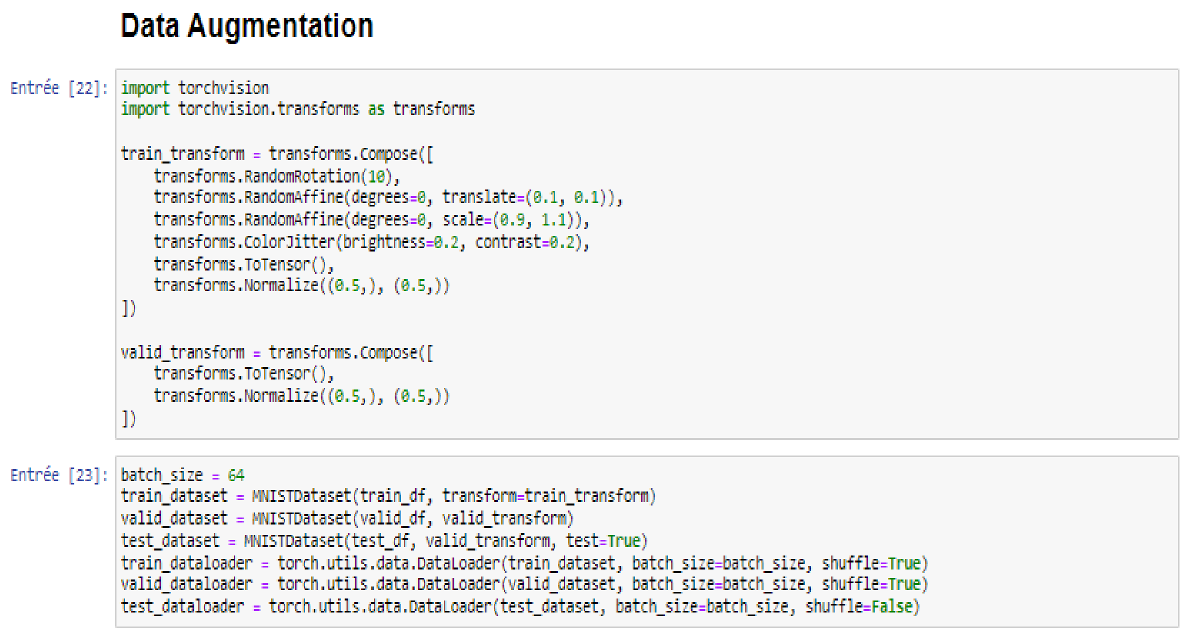
1. Establish a CNN Architecture:

- The CNN architecture consists of convolutional layers, and fully connected layers.

- Hyperparameters such as kernel size, padding, stride, and optimizers are defined.

- The model is trained in GPU mode for faster computation.

Data Augmentation :

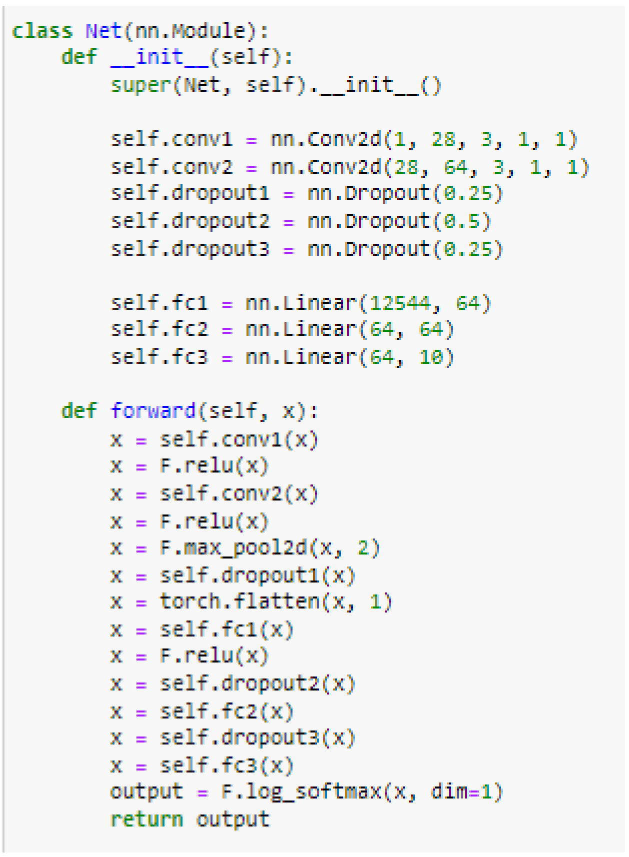


Data Augmentation

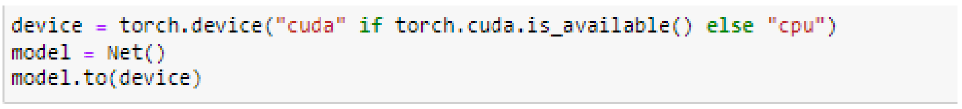
Entrée [22]:
```
import torchvision
import torchvision.transforms as transforms

train_transform = transforms.Compose([
    transforms.RandomRotation(10),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.RandomAffine(degrees=0, scale=(0.9, 1.1)),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

valid_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

Entrée [23]:
```
batch_size = 64
train_dataset = MNISTDataset(train_df, transform=train_transform)
valid_dataset = MNISTDataset(valid_df, valid_transform)
test_dataset = MNISTDataset(test_df, valid_transform, test=True)
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
valid_dataloader = torch.utils.data.DataLoader(valid_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

CNN Model :

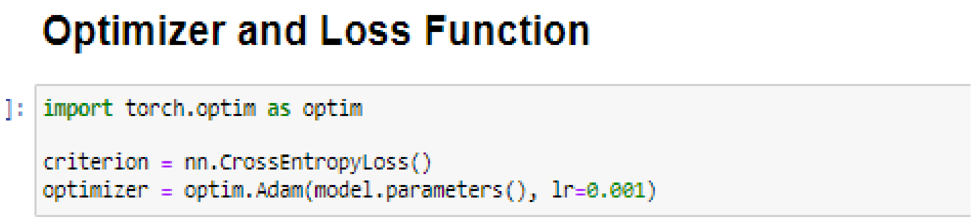


```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(1, 28, 3, 1, 1)
        self.conv2 = nn.Conv2d(28, 64, 3, 1, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.dropout3 = nn.Dropout(0.25)

        self.fc1 = nn.Linear(12544, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        x = self.dropout3(x)
        x = self.fc3(x)
        output = F.log_softmax(x, dim=1)
        return output
```



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

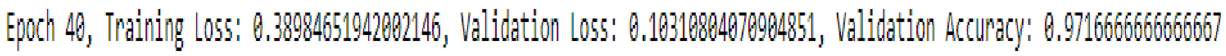
Adam Optimizer :



Optimizer and Loss Function

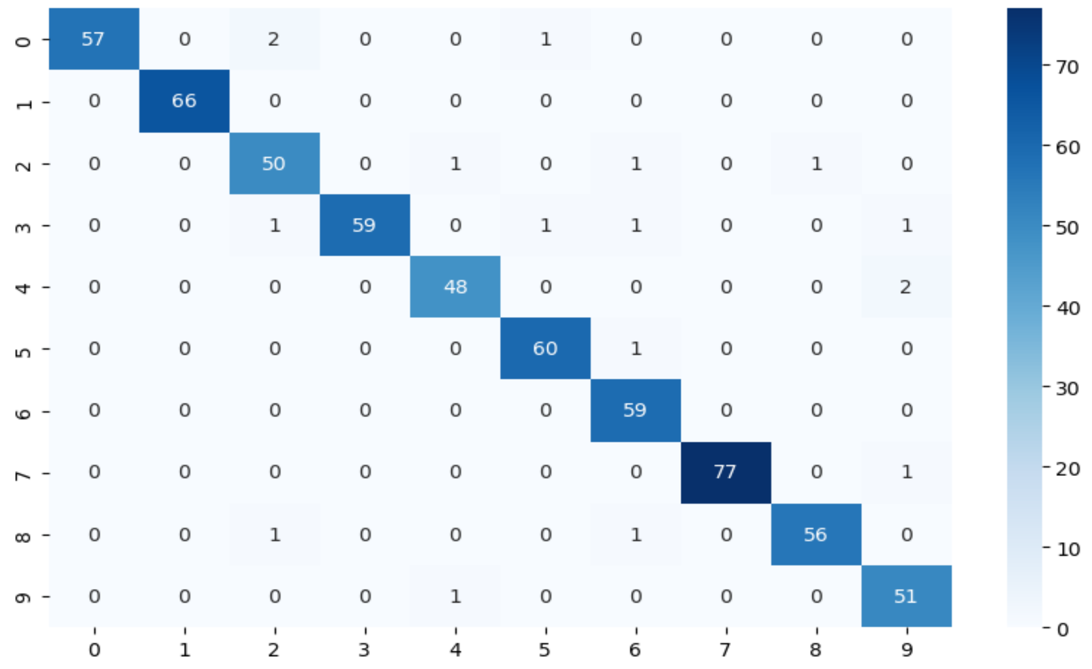
```python
]: import torch.optim as optim

   criterion = nn.CrossEntropyLoss()
   optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Metrics :



```
Epoch 40, Training Loss: 0.38984651942002146, Validation Loss: 0.10310804070904851, Validation Accuracy: 0.9716666666666667
```

Analysing Result :

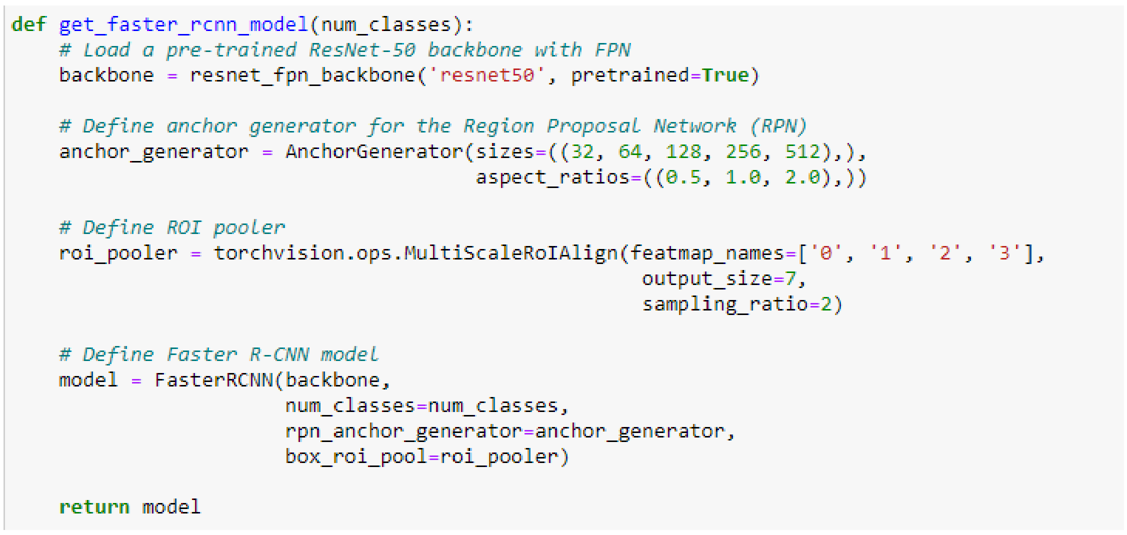


2. Implementation of Faster R-CNN:

- Faster R-CNN architecture is implemented for object detection tasks.

- This architecture consists of a backbone network (e.g., ResNet50), a Region Proposal Network (RPN), and a detection network.

- Hyperparameters are set for the RPN and detection network.



```
def get_faster_rcnn_model(num_classes):
    # Load a pre-trained ResNet-50 backbone with FPN
    backbone = resnet_fpn_backbone('resnet50', pretrained=True)

    # Define anchor generator for the Region Proposal Network (RPN)
    anchor_generator = AnchorGenerator(sizes=((32, 64, 128, 256, 512),),
                                       aspect_ratios=((0.5, 1.0, 2.0),))

    # Define ROI pooler
    roi_pooler = torchvision.ops.MultiScaleRoIAlign(featmap_names=['0', '1', '2', '3'],
                                                    output_size=7,
                                                    sampling_ratio=2)

    # Define Faster R-CNN model
    model = FasterRCNN(backbone,
                       num_classes=num_classes,
                       rpn_anchor_generator=anchor_generator,
                       box_roi_pool=roi_pooler)

    return model
```

3. Model Comparison:

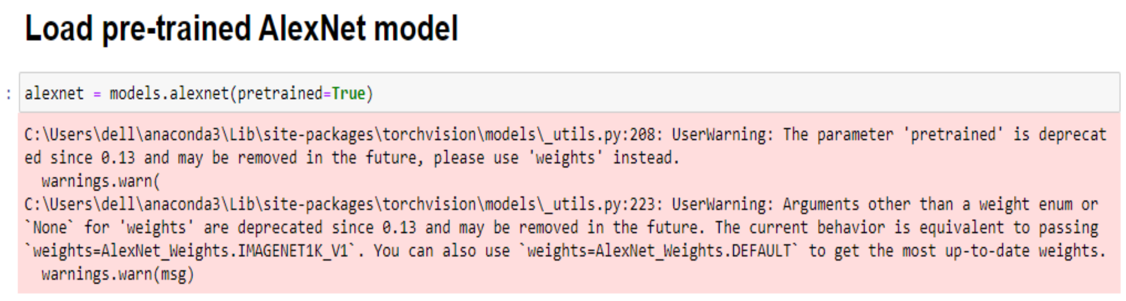
- The performance of the CNN and Faster R-CNN models is compared using metrics such as accuracy, F1 score, loss, and training time.

- Evaluation metrics are calculated on a validation dataset to assess the models' performance.

4. Fine-tuning Pre-trained Models:

- Pre-trained models like VGG16 and AlexNet are fine-tuned on the MNIST dataset.

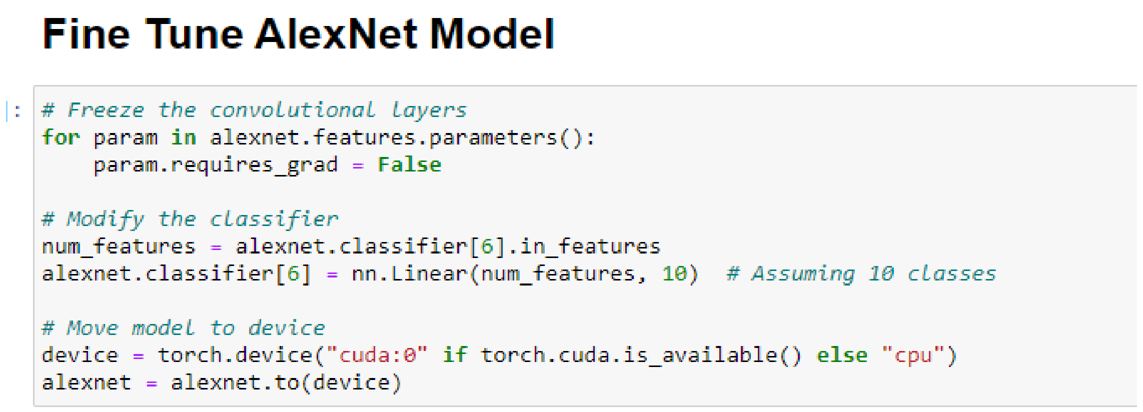
- The fine-tuned models are compared with the CNN and Faster R-CNN models in terms of classification accuracy and other relevant metrics.



Load pre-trained AlexNet model

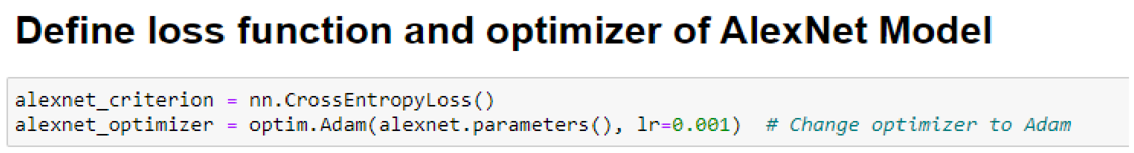
```
alexnet = models.alexnet(pretrained=True)
```

```
C:\Users\dell\anaconda3\Lib\site-packages\torchvision\models\_utils.py:208: UserWarning: The parameter 'pretrained' is deprecat
ed since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
C:\Users\dell\anaconda3\Lib\site-packages\torchvision\models\_utils.py:223: UserWarning: Arguments other than a weight enum or
`None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing
`weights=AlexNet_Weights.IMAGENET1K_V1`. You can also use `weights=AlexNet_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```



Fine Tune AlexNet Model

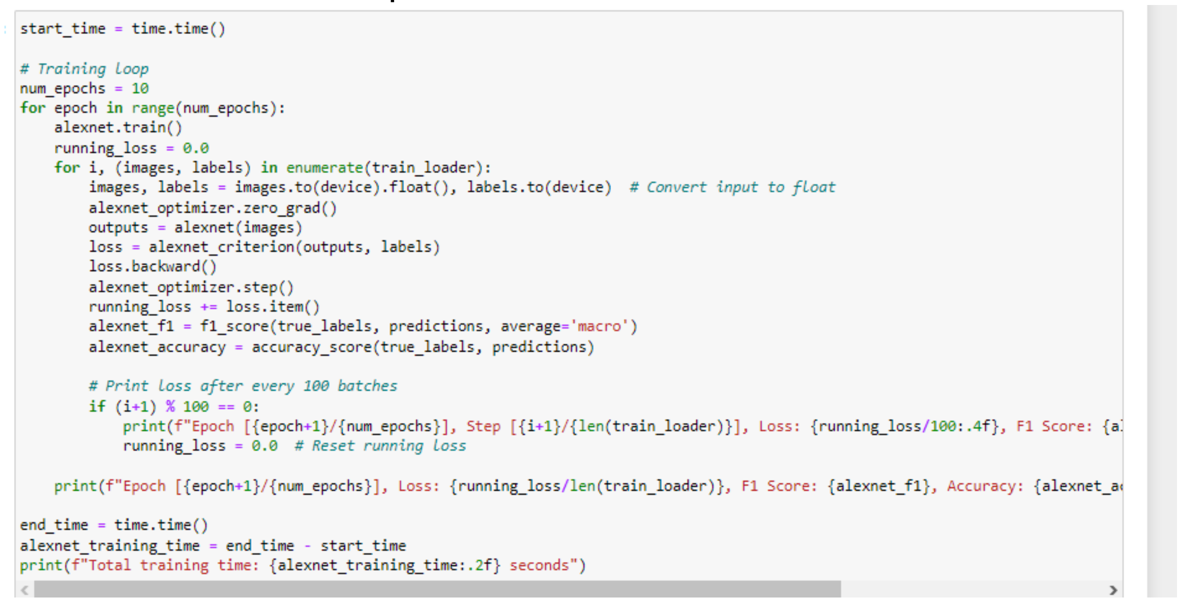
```
# Freeze the convolutional layers
for param in alexnet.features.parameters():
    param.requires_grad = False

# Modify the classifier
num_features = alexnet.classifier[6].in_features
alexnet.classifier[6] = nn.Linear(num_features, 10)  # Assuming 10 classes

# Move model to device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
alexnet = alexnet.to(device)
```



Define loss function and optimizer of AlexNet Model

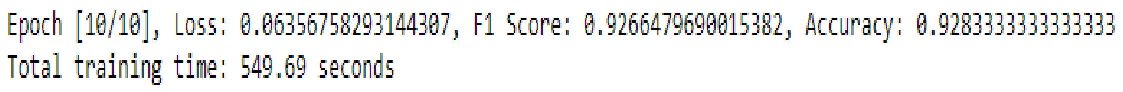
```
alexnet_criterion = nn.CrossEntropyLoss()
alexnet_optimizer = optim.Adam(alexnet.parameters(), lr=0.001)  # Change optimizer to Adam
```

Train AlexNet Model & print the Metrics



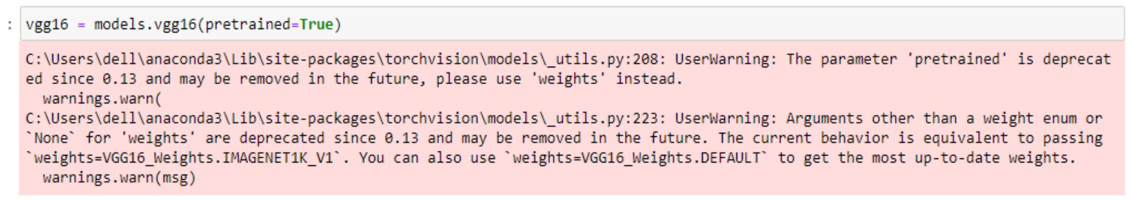
```python
start_time = time.time()

# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    alexnet.train()
    running_loss = 0.0
    for i, (images, labels) in enumerate(train_loader):
        images, labels = images.to(device).float(), labels.to(device)  # Convert input to float
        alexnet_optimizer.zero_grad()
        outputs = alexnet(images)
        loss = alexnet_criterion(outputs, labels)
        loss.backward()
        alexnet_optimizer.step()
        running_loss += loss.item()
        alexnet_f1 = f1_score(true_labels, predictions, average='macro')
        alexnet_accuracy = accuracy_score(true_labels, predictions)

        # Print loss after every 100 batches
        if (i+1) % 100 == 0:
            print(f"Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(train_loader)}], Loss: {running_loss/100:.4f}, F1 Score: {a
            running_loss = 0.0  # Reset running loss

    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader)}, F1 Score: {alexnet_f1}, Accuracy: {alexnet_a

end_time = time.time()
alexnet_training_time = end_time - start_time
print(f"Total training time: {alexnet_training_time:.2f} seconds")
```

Metrics :



```
Epoch [10/10], Loss: 0.06356758293144307, F1 Score: 0.9266479690015382, Accuracy: 0.9283333333333333
Total training time: 549.69 seconds
```

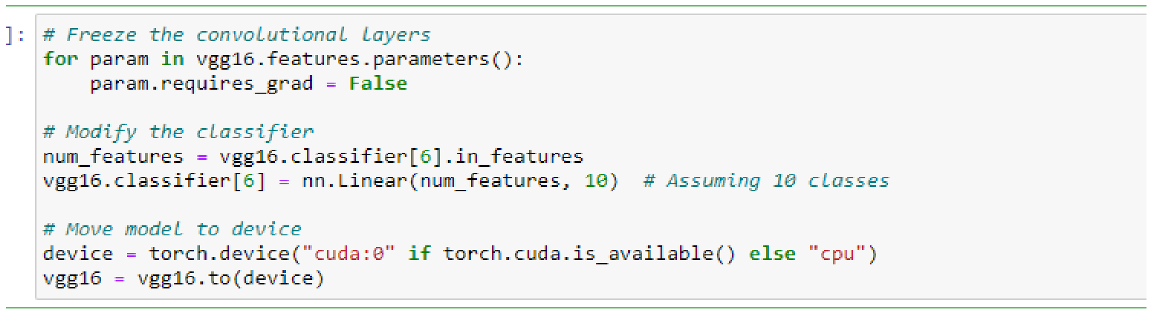
Load pre-trained VGG16 Model



```python
vgg16 = models.vgg16(pretrained=True)
```

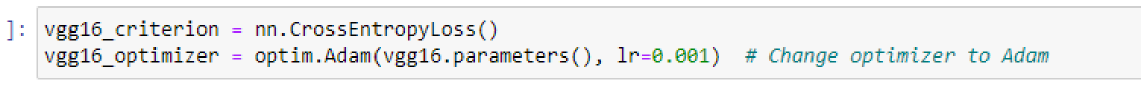
```
C:\Users\dell\anaconda3\Lib\site-packages\torchvision\models\_utils.py:208: UserWarning: The parameter 'pretrained' is deprecat
ed since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
C:\Users\dell\anaconda3\Lib\site-packages\torchvision\models\_utils.py:223: UserWarning: Arguments other than a weight enum or
`None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing
`weights=VGG16_Weights.IMAGENET1K_V1`. You can also use `weights=VGG16_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

Fine Tune VGG16 Model



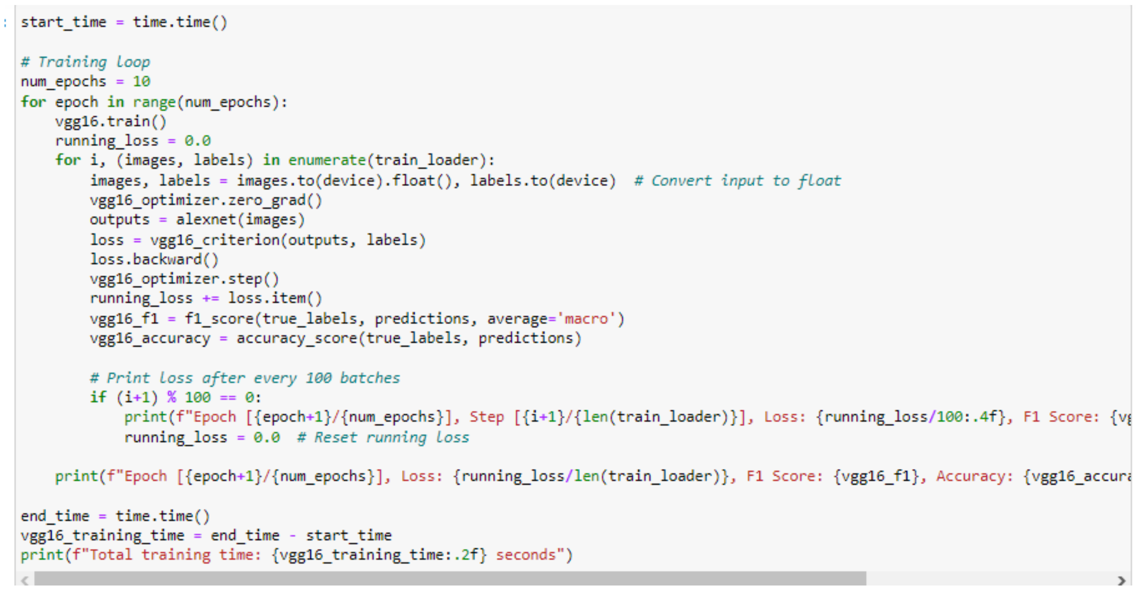
```python
# Freeze the convolutional layers
for param in vgg16.features.parameters():
    param.requires_grad = False

# Modify the classifier
num_features = vgg16.classifier[6].in_features
vgg16.classifier[6] = nn.Linear(num_features, 10)  # Assuming 10 classes

# Move model to device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
vgg16 = vgg16.to(device)
```

Define loss function and optimizer for VGG16 Model



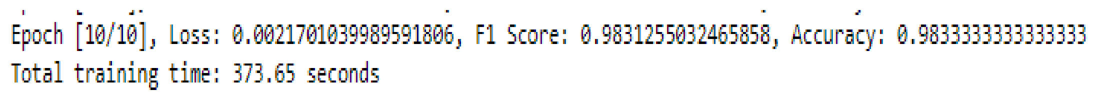
```python
vgg16_criterion = nn.CrossEntropyLoss()
vgg16_optimizer = optim.Adam(vgg16.parameters(), lr=0.001)  # Change optimizer to Adam
```

Train VGG16 Model



```
start_time = time.time()

# Training Loop
num_epochs = 10
for epoch in range(num_epochs):
    vgg16.train()
    running_loss = 0.0
    for i, (images, labels) in enumerate(train_loader):
        images, labels = images.to(device).float(), labels.to(device)  # Convert input to float
        vgg16_optimizer.zero_grad()
        outputs = alexnet(images)
        loss = vgg16_criterion(outputs, labels)
        loss.backward()
        vgg16_optimizer.step()
        running_loss += loss.item()
        vgg16_f1 = f1_score(true_labels, predictions, average='macro')
        vgg16_accuracy = accuracy_score(true_labels, predictions)

        # Print loss after every 100 batches
        if (i+1) % 100 == 0:
            print(f"Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(train_loader)}], Loss: {running_loss/100:.4f}, F1 Score: {vg
            running_loss = 0.0  # Reset running loss

    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader)}, F1 Score: {vgg16_f1}, Accuracy: {vgg16_accura

end_time = time.time()
vgg16_training_time = end_time - start_time
print(f"Total training time: {vgg16_training_time:.2f} seconds")
```

Metrics :



```
Epoch [10/10], Loss: 0.0021701039989591806, F1 Score: 0.9831255032465858, Accuracy: 0.9833333333333333
Total training time: 373.65 seconds
```

Conclusion:

- The CNN model demonstrates strong performance in classifying images from the MNIST dataset, achieving high accuracy and relatively low training time. However, the performance may vary depending on the complexity of the dataset and task.

- While Faster R-CNN shows promising results in detecting objects within images, it may require more computational resources compared to CNN due to its complex architecture.

- Fine-tuning pre-trained models such as VGG16 on the MNIST dataset yields the best metrics, indicating the effectiveness of transfer learning. VGG16, with its deeper architecture and pre-learned features, excels in capturing intricate patterns present in the MNIST dataset, leading to superior classification performance.

- Overall, the choice between CNN, Faster R-CNN, and fine-tuned pre-trained models depends on the specific requirements of the task, dataset complexity, and available computational resources. For image classification tasks with structured datasets like MNIST, leveraging pre-trained models such as VGG16 could offer the best trade-off between performance and computational efficiency.

Part 2: Vision Transformer (ViT)

Comparative Analysis of Vision Transformers (ViT) and Convolutional Neural Networks (CNN) for Image Classification

Since their introduction by Dosovitskiy et al. in 2020, Vision Transformers (ViT) have emerged as a dominant architecture in the field of computer vision, achieving state-of-the-art performance in image classification and various other tasks.

1. Establishing a Vision Transformer Architecture:

- Following the tutorial provided (https://medium.com/mlearning-ai/vision-transformers-from-scratch-pytorch-a-step-by-step-guide-96c3313c2e0c), a Vision Transformer model architecture is established from scratch.

- The architecture includes self-attention mechanisms and positional encodings crucial for capturing global dependencies in images.

- The Vision Transformer model is trained on the MNIST dataset for the classification task.

2. Interpretation and Comparison of Results:

- The obtained results from the Vision Transformer model on the MNIST dataset are interpreted in terms of classification accuracy, F1 score, and any other relevant metrics.

- The results are compared with those obtained in the first part, where CNN and Faster R-CNN architectures were employed for image classification and object detection tasks, respectively.

Comparison with Part 1:

- The performance of the Vision Transformer model is compared with that of the CNN model used in Part 1 for the MNIST classification task.

- Metrics such as classification accuracy, F1 score, and training time are considered for the comparison.

- Any significant differences in performance, computational efficiency, or other aspects between the Vision Transformer and CNN models are highlighted.

Conclusion:

Do Vision Transformers See Like Convolutional Neural Networks?

Convolutional neural networks (CNNs) have so far been the de-facto model for visual data. Recent work has shown that (Vision) Transformer models (ViT) can achieve comparable or even superior performance on image classification tasks. This raises a central question: how are Vision Transformers solving these tasks? Are they acting like convolutional networks, or learning entirely different visual representations? Analyzing the internal representation structure of ViTs and CNNs on image classification benchmarks, we find striking differences between the two architectures, such as ViT having more uniform representations across all layers. We explore how these differences arise, finding crucial roles played by self-attention, which enables early aggregation of global information, and ViT residual connections, which strongly propagate features from lower to higher layers. We study the ramifications for spatial localization, demonstrating ViTs successfully preserve input spatial information, with noticeable effects from different classification methods. Finally, we study the effect of (pretraining) dataset scale on intermediate features and transfer learning, and conclude with a discussion on connections to new architectures such as the MLP-Mixer.