Aim

To develop a robust and efficient classifier for recognizing hand gestures (rock, paper, and scissors) using the FasterViT model, leveraging the computational power of an RTX 3060 GPU.

Objectives

1. Preprocess and augment the dataset to enhance model performance.
2. Implement and fine-tune the FasterViT model for hand gesture classification.
3. Evaluate the model's accuracy and performance on the validation dataset.
4. Deploy the model to make real-time predictions on new images.

Summary

This project involves building a rock-paper-scissors classifier using the FasterViT architecture. The dataset is preprocessed and augmented to improve model generalization. The model is then trained and evaluated, achieving high accuracy. Finally, the trained model is used to predict hand gestures on new images, demonstrating its practical application.

Tools and Libraries Used

- Python
- PyTorch
- torchvision
- FasterViT (custom implementation)
- Jupyter Lab
- NVIDIA RTX 3060 GPU

Procedure

1. **Data Preprocessing and Augmentation:**

CODE:

```
from torchvision import datasets, transforms


data_dir = 'RockPaperScissorsDataset'

data_transforms = {

   'train': transforms.Compose([

      transforms.RandomResizedCrop(224),

      transforms.RandomHorizontalFlip(),
```

```
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), data_transforms[x]) for x in ['train', 'val']}
dataloaders = {x: DataLoader(image_datasets[x], batch_size=32, shuffle=True, num_workers=4) for x in ['train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
class_names = image_datasets['train'].classes
```

2. **Model Initialization:**

CODE:

```
from fastervit import create_model

model = create_model('faster_vit_0_224', pretrained=True, model_path="tmp/faster_vit_0.pth.tar")
num_ftrs = model.head.in_features
model.head = torch.nn.Linear(num_ftrs, len(class_names))
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = model.to(device)
```

3. **Training Setup:**

CODE:

```
import torch.optim as optim
```

```
from torch.optim import lr_scheduler


criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
```

4. **Model Training:**

CODE:

```
def train_model(model, criterion, optimizer, scheduler, num_epochs=5):
    since = time.time()
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

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

        for phase in ['train', 'val']:
            model.train() if phase == 'train' else model.eval()
            running_loss = 0.0
            running_corrects = 0

            for inputs, labels in dataloaders[phase]:
                inputs, labels = inputs.to(device), labels.to(device)
                optimizer.zero_grad()

                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)
```

```
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

            print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

    time_elapsed = time.time() - since
    print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s')
    print(f'Best val Acc: {best_acc:.4f}')
    model.load_state_dict(best_model_wts)
    return model

model = train_model(model, criterion, optimizer, exp_lr_scheduler, num_epochs=5)
torch.save(model.state_dict(), 'faster_vit_custom_model.pth')
```

5. **Model Evaluation and Prediction:**

CODE:

```
from PIL import Image
from fastervit import create_model

model = create_model('faster_vit_0_224', pretrained=False)
model.head = torch.nn.Linear(model.head.in_features, len(class_names))
model.load_state_dict(torch.load('faster_vit_custom_model.pth'))
model.eval()

preprocess = transforms.Compose([
   transforms.Resize(256),
   transforms.CenterCrop(224),
   transforms.ToTensor(),
   transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

def load_image(image_path):
   image = Image.open(image_path).convert('RGB')
   image = preprocess(image).unsqueeze(0)
   return image.to(device)

def predict(image_path, model, class_names):
   image = load_image(image_path)
   with torch.no_grad():
      outputs = model(image)
      _, preds = torch.max(outputs, 1)
      return class_names[preds.item()]

class_names = ['paper', 'rock', 'scissors']
```

```
image_path = 'RockPaperScissorsDataset/test/scissors/scissors-
hires1_png.rf.b7f86aab3e36b747b67db8fa7368bc17.jpg'

predicted_class = predict(image_path, model, class_names)

print(predicted_class)
```

6. **Drawing Predictions on Images:**

CODE:

```
from PIL import Image, ImageDraw, ImageFont


def predict_and_draw(image_path, model, class_names):
    image = Image.open(image_path).convert('RGB')
    input_tensor = preprocess(image).unsqueeze(0).to(device)


    with torch.no_grad():
        outputs = model(input_tensor)
        _, preds = torch.max(outputs, 1)
        predicted_class = class_names[preds.item()]


    draw = ImageDraw.Draw(image)
    font = ImageFont.load_default()
    text = f'Predicted: {predicted_class}'
    text_bbox = draw.textbbox((10, 10), text, font=font)
    text_width, text_height = text_bbox[2] - text_bbox[0], text_bbox[3] - text_bbox[1]
    text_position = (10, 10)
    draw.rectangle([text_position, (text_position[0] + text_width, text_position[1] + text_height)],
fill="black")
    draw.text(text_position, text, fill="white", font=font)


    image.show()
    output_image_path = "output_with_prediction.jpg"
```

```
    image.save(output_image_path)
    print(f"Saved output image with prediction: {output_image_path}")


predict_and_draw(image_path, model, class_names)
```

Highlights

1. **Data Augmentation:** Applied random resizing, cropping, and horizontal flipping to improve model robustness.
2. **Model Fine-tuning:** Modified the final layer of the FasterViT model to match the number of classes in the custom dataset.
3. **Training Efficiency:** Leveraged GPU acceleration and an efficient learning rate scheduler for optimized training.
4. **Real-time Prediction:** Implemented a pipeline for loading, preprocessing, and predicting new images with the trained model.

Conclusion

The project successfully demonstrates the use of the FasterViT model for classifying hand gestures in a rock-paper-scissors game. By leveraging data augmentation techniques and fine-tuning a pre-trained model, we achieved high accuracy and efficiency. The model can be further enhanced with additional data and more sophisticated augmentation techniques. The approach showcases the potential of state-of-the-art vision transformers in real-time image classification tasks.