به نام خدا

دانشگاه تهران - دانشگاه صنعتی خواجه نصیرالدین طوسی تهران

دانشکده مهندسی برق و کامپیوتر

ویژن ترنسفورمرها

نام و نام خانوادگی	محمدجواد احمدی
شمارهٔ دانشجویی	۴۰۱۰۰۰۸۶

فهرست مطالب

فهرست تصاویر

پرسش ۲. استفاده از ویژن‌ترنسفورمرها برای طبقه‌بندی تصاویر

۱ پاسخ پرسش دوم

توضیح پوشهٔ کدهای استفاده از Vision Transformer برای طبقه‌بندی تصاویر

کدهای مربوط به این قسمت، علاوه بر پوشهٔ محلی کدها در این لینک گوگل‌کولب آورده شده است.

۱.۱ پاسخ قسمت ۱ - لودکردن دیتاست و انجام پیش‌پردازش‌های لازم

در مقاله در مورد مجموعه‌داده این‌گونه توضیح داده شده است که برای تنظیم دقیق مدل‌ها که تنها روی مجموعه‌دادهٔ Imagenet-1K آموزش دیده‌اند، مجموعه‌دادهٔ CIFAR-10 انتخاب شده است. مجموعه‌دادهٔ Imagenet-1K شامل ۱۰۰۰ کلاس با ۱.۲ میلیون نمونه با وضوح بالا است. مجموعه‌دادهٔ CIFAR-10 که تصاویر آن جزئیات چندانی ندارد، شامل ۶۰۰۰۰ تصویر رنگی ۳۲ در ۳۲ است که به ده کلاس تقسیم شده‌اند و هر کلاس شامل ۶۰۰۰ تصویر است. برای آموزش از ۵۰۰۰۰ تصویر و برای آزمون از ۱۰۰۰۰ تصویر استفاده می‌شود. هنگام ورودی‌دهی، تصاویر ۳۲ در ۳۲ را به ابعاد ۲۲۴ در ۲۲۴ افزایش اندازه داده‌اند. مقاله سعی می‌کند ببیند که آیا مدل‌ها همچنان با اعمال درون‌یابی دوخطی ساده به تصاویر با وضوح پایین عملکرد خوبی دارند یا خیر.

با این توضیحات، دستوراتی را برای نصب کتابخانه‌های مورد نیاز، بارگیری و پیش‌پردازش مجموعه‌دادهٔ CIFAR-10 و ذخیرهٔ نمونه‌های با وضوح بالا از آن مجموعه‌داده در یک مسیر جدید می‌نویسیم. ابتدا کتابخانه‌های موردنیاز را نصب و فراخوانی می‌کنیم. سپس دستوری را برای تعریف تبدیلی به نام `upscale_transform` استفاده می‌کنیم که تصاویر را با روش مطرح‌شده در مقاله به ابعاد 224*224 افزایش اندازه می‌دهد. در ادامه، مجموعه‌دادهٔ CIFAR-10 را دریافت کرده و با استفاده از تبدیل مذکور، تصاویر را به ابعاد گفته‌شده تغییر اندازه می‌دهیم. `train=True` و `train=False` به‌ترتیب برای بارگیری مجموعه‌دادهٔ آموزشی و آزمون استفاده می‌شوند. در ادامه یک مسیر جدید برای ذخیره‌سازی داده‌های پیشپردازش‌شده تعیین می‌کنیم و آن‌ها را به‌صورتی مناسب که قابل بارگیری باشد در آن مسیر ذخیره می‌کنیم. دستورات به شرح زیر است:

```
# Install the required libraries
!pip install torch torchvision datasets transformers

import os
import torch
import torchvision
import torchvision.transforms as transforms

# Define the transformations
upscale_transform = transforms.Resize((224, 224), interpolation=transforms.InterpolationMode.
    BILINEAR)

```

```python
# Download and load the CIFAR-10 dataset
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, transform=
    upscale_transform, download=True)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, transform=
    upscale_transform, download=True)

# Specify the new root directory for the resized dataset
new_root = './resized_data'

# Create the new directory if it doesn't exist
if not os.path.exists(new_root):
    os.makedirs(new_root)

# Save the resized dataset in the new root directory
torch.save(train_dataset, os.path.join(new_root, 'train_dataset.pth'))
torch.save(test_dataset, os.path.join(new_root, 'test_dataset.pth'))
```

۲.۱ پاسخ قسمت ۲ - شبکهٔ کانولوشنی

برای هدف این سوال ابتدا تبدیل‌های مورد نیاز برای تصاویر را تعریف می‌کنیم تا فرآیند فراخوانی و پیش‌پردازش داده‌ها شفاف باشند و از فایل‌های ذخیره‌شده از قسمت قبل استفاده نکنیم. مطابق مقاله، تبدیل‌های مختلفی می‌توانند اعمال شوند، مانند تغییر اندازه (resize)، برگرداندن افقی (horizontal flip) و تبدیل تصویر به تنسور (convert to tensor). تمام این تبدیل‌ها در کلاس `transforms.Compose` به صورت زنجیره‌ای اعمال می‌شوند. در این حالت، تبدیلات عبارتند از: تغییر اندازهٔ تصویر به ابعاد 224*224 با استفاده از روش دوخطی (bilinear interpolation)، اعمال برگرداندن افقی برای افزایش تنوع داده‌ها.، و تبدیل تصویر به تنسور برای استفاده در مدل‌های پیچشی. در ادامه، مجموعه‌دادهٔ CIFAR-10 را دانلود و بارگیری می‌کنیم. دستهٔ آموزشی و آزمون را به‌صورت مجزا تعریف می‌کنیم. تبدیل‌های تعریف‌شده در مرحلهٔ قبلی با استفاده از پارامتر `transform` به هر دو مجموعه‌داده اعمال می‌شوند. مکان ذخیره‌سازی داده‌ها در پوشه data/. تعیین شده است. اگر داده‌ها قبلاً دانلود شده باشند، مجدداً دانلود نمی‌شوند. سپس، داده‌های آموزش و آزمون به عنوان دسته‌های مجزا تعریف می‌شوند. با استفاده از `torch.utils.data.DataLoader`، داده‌ها در دسته‌های مشخص (در این‌جا به تعداد ۶۴ تصویر در هر دسته) بارگیری و آمادهٔ استفاده در آموزش و ارزیابی مدل می‌شوند. داده‌های آموزش تصادفی مخلوط می‌شوند (`shuffle=True`)، اما داده‌های آزمون ترتیب خود را حفظ می‌کنند (`shuffle=False`).

در ادامه و مرحلهٔ بعد، دستوراتی را برای بارگیری مدل پیش‌آموزش‌دیدهٔ VGG-19 می‌نویسیم و برای این منظور با استفاده از کتابخانهٔ `torchvision.models`، مدل VGG-19 را با پارامتر `pretrained=True` بارگیری می‌کنیم. در ادامه، مطابق مقاله، لایه‌های مدل از یک لایهٔ خاص به بعد را آزادسازی می‌کنیم. بر اساس مقاله، لایهٔ "`block5_conv1`" از مدل VGG-19 به‌عنوان لایهٔ آغازین آزادسازی انتخاب شده است. ابتدا تمام پارامترهای لایه‌ها را با مقدار پیش‌فرض `requires_grad=False` مشخص می‌کنیم. سپس با استفاده از حلقهٔ `for` و بررسی نام و پارامترهای لایه‌ها، پارامترهای لایه‌ها را آزاد می‌کنیم تا قابل به‌روزرسانی باشند. از متغیر `unfreeze` برای آزادسازی پارامترها از لایه مورد نظر به بعد استفاده می‌کنیم.

در ادامه دستوراتی را برای تدقیق (Fine-tuning) مدل و تعیین تابع هدف و بهینه‌ساز استفاده می‌نویسیم. در ابتدای امر، از کلاس `torch.nn.CrossEntropyLoss` برای محاسبه تابع خطا (loss) در مسألهٔ دسته‌بندی استفاده می‌کنیم. این تابع به

صورت پیش‌فرض برای مسائل دسته‌بندی چند دسته‌ای مناسب است. سپس از کلاس `torch.optim.Adam` استفاده می‌کنیم و با استفاده از `filter(lambda p: p.requires_grad, model.parameters())`، فقط پارامترهایی که `requires_grad=True` دارند (یعنی پارامترهای آزاد برای به‌روزرسانی) را انتخاب می‌کنیم. در ادامه، دستگاه محاسباتی برای اجرای مدل را تعیین می‌کنیم. این دستور به صورتی نوشته شده که اگر دستگاه GPU (CUDA) در دسترس باشد، مدل را روی GPU اجرا می‌کند (`device = torch.device("cuda")`)، در غیر این صورت از پردازندهٔ مرکزی استفاده می‌کند (`device = torch.device("cpu")`). این کار با استفاده از `torch.cuda.is_available` انجام می‌شود. با انجام این مراحل، تابع هدف و بهینه‌ساز تعریف می‌شوند و مدل نیز به دستگاه محاسباتی منتقل می‌شود. این مراحل معمولاً قبل از شروع آموزش مدل به‌کار می‌روند. در ادامه، پارامترهای مربوط به آموزش را تعریف می‌کنیم که شامل تعداد دوره‌های آموزش (`num_epochs`)، ضریب کاهش نرخ یادگیری (`lr_factor`)، میزان تحمل (`lr_patience`)، و حداقل نرخ یادگیری (`min_lr`) می‌شود. هم‌چنین لیست‌هایی را برای ذخیرهٔ معیارهای آموزش و ارزیابی مدل تعریف می‌کنیم که شامل لیست خطاهای آموزش (`train_losses`)، دقت آموزش (`train_accs`)، خطاهای ارزیابی (`val_losses`) و دقت ارزیابی (`val_accs`) است.

در انتها از کلاس `torch.optim.lr_scheduler.ReduceLROnPlateau` استفاده می‌کنیم و تمام تنظیمات مدنظر را اِعمال می‌کنیم. با انجام این فرآیند، حلقهٔ دوره‌های آموزش را تعریف می‌کنیم و با استفاده از `range(num_epochs)`، دوره‌های آموزش را ایجاد می‌کنیم. در هر دوره، مدل در حال آموزش قرار می‌گیرد (`model.train`) و خطا و دقت آموزش برای داده‌های آموزش محاسبه می‌شود. سپس مدل در حال ارزیابی قرار می‌گیرد (`model.eval`) و خطا و دقت ارزیابی برای داده‌های ارزیابی محاسبه می‌شود. سپس معیارهای محاسبه شده به لیست‌های مربوطه اضافه می‌شوند. در انتها نرخ یادگیری به عنوان یک پارامتر در بهینه‌ساز به‌روزرسانی می‌شود (`lr_scheduler.step(val_acc)`) و مقدار فعلی آن در هر چرخه نمایش داده می‌شود. در بخش بررسی بهبود عملکرد، عملکرد مدل با مدل بهتر قبلی مقایسه می‌شود. اگر دقت ارزیابی بهتر باشد، مدل بهتر به‌روزرسانی می‌شود و تعداد دوره‌های بدون بهبود صفر می‌شود. اگر دقت ارزیابی بهبود نیابد، تعداد دوره‌های بدون بهبود افزایش می‌یابد و اگر این تعداد به حد صبر تعیینی برسد، آموزش متوقف می‌شود. با انجام این مراحل، مدل به مدت تعداد دوره‌های آموزش مشخص شده آموزش می‌بیند و در هر دوره، خطا و دقت آموزش و ارزیابی ثبت می‌شود. همچنین، نرخ یادگیری به‌روزرسانی می‌شود و به‌روزرسانی بهبود عملکرد نیز انجام می‌شود. پس از انجام این کارها دستوراتی را می‌نویسیم تا نمودارها و معیارهای مختلفی را برای ارزیابی عملکرد مدل نمایش دهیم. دستورات به صورتی است که در برنامهٔ ۲ آورده شده است.

Program 1: VGG-19 Implementation

```
import torch
import torchvision
import torchvision.transforms as transforms
import torchvision.models as models
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import numpy as np
from sklearn.metrics import f1_score, recall_score, accuracy_score, precision_score
import warnings
warnings.filterwarnings('ignore')

# Define the transformations
upscale_transform = transforms.Compose([
    transforms.Resize((224, 224), interpolation=transforms.InterpolationMode.BILINEAR),
```

```
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor()  # Add this line to convert images to tensors
])

# Download and load the CIFAR-10 dataset
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, transform=
    upscale_transform, download=True)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, transform=
    upscale_transform, download=True)

# Define the dataloaders
batch_size = 64
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Load the pre-trained VGG-19 model
model = models.vgg19(pretrained=True)

# Unfreeze layers starting from 'block5_conv1'
unfreeze_from_layer = 'block5_conv1'
unfreeze = False

for name, param in model.named_parameters():
    if unfreeze:
        param.requires_grad = True
    if name == unfreeze_from_layer:
        unfreeze = True

# Fine-tune the model
criterion = torch.nn.CrossEntropyLoss()
initial_lr = 0.0001
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=initial_lr
    )

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

num_epochs = 20
lr_factor = 0.6
lr_patience = 1
min_lr = 0.0000001

train_losses = []
train_accs = []
val_losses = []
```

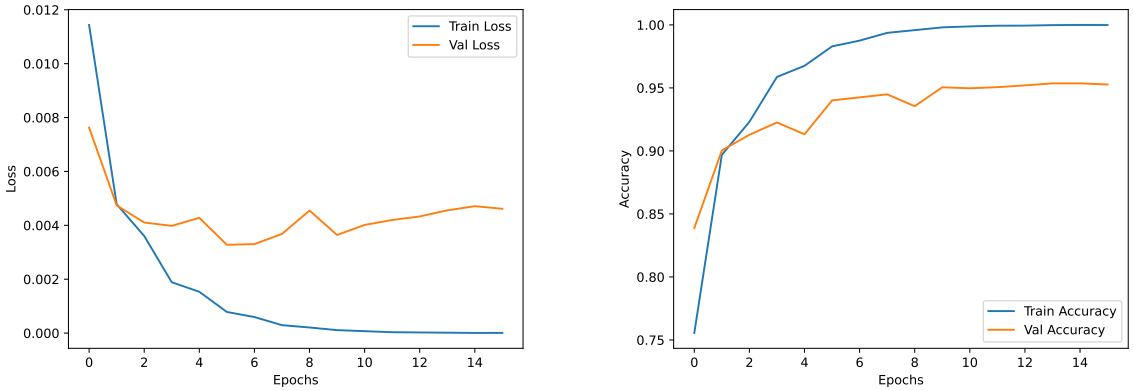
```
val_accs = []
lr_values = []

best_val_acc = 0.0
stop_after_epochs = 2
epochs_without_improvement = 0
lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, factor=lr_factor, patience=
    lr_patience,
                                                          min_lr=min_lr, verbose=True)

for epoch in range(num_epochs):
    model.train()
    train_loss = 0.0
    train_acc = 0.0

    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        train_acc += (predicted == labels).sum().item()

    train_loss /= len(train_loader.dataset)
    train_acc /= len(train_loader.dataset)

    train_losses.append(train_loss)
    train_accs.append(train_acc)

    model.eval()
    val_loss = 0.0
    val_acc = 0.0
    y_true = []
    y_pred = []

    with torch.no_grad():
        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)
```

```
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            val_acc += (predicted == labels).sum().item()

            y_true.extend(labels.tolist())
            y_pred.extend(predicted.tolist())

    val_loss /= len(test_loader.dataset)
    val_acc /= len(test_loader.dataset)

    val_losses.append(val_loss)
    val_accs.append(val_acc)

    lr_values.append(optimizer.param_groups[0]['lr'])
    lr_scheduler.step(val_acc)

    print(f"Epoch {epoch + 1}/{num_epochs}: Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}, "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}, LR: {optimizer.param_groups[0]['lr']:.8f}")

    if val_acc > best_val_acc:
        best_val_acc = val_acc
        epochs_without_improvement = 0
    else:
        epochs_without_improvement += 1
        if epochs_without_improvement >= stop_after_epochs:
            print(f"Validation accuracy has not improved for {stop_after_epochs} epochs. "
                  f"Stopping training...")
            break

# Plot val/train accuracy and loss
plt.figure()
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Val Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.savefig('lossplot22.pdf')
plt.show()

plt.figure()
```

```
plt.plot(train_accs, label='Train Accuracy')
plt.plot(val_accs, label='Val Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.savefig('accuracyplot22.pdf')
plt.show()

# Calculate and plot confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues', cbar=False)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.savefig('confusionmatrix22.pdf')
plt.show()

# Print F1-score, recall, accuracy, and precision for all classes
f1_scores = f1_score(y_true, y_pred, average=None)
recall_scores = recall_score(y_true, y_pred, average=None)
accuracy = accuracy_score(y_true, y_pred)
precision_scores = precision_score(y_true, y_pred, average=None)

for i in range(len(f1_scores)):
    print(f"Class {i}: F1-Score: {f1_scores[i]:.4f}, Recall: {recall_scores[i]:.4f}, "
          f"Precision: {precision_scores[i]:.4f}")

print(f"Overall Accuracy: {accuracy:.4f}")
```

نتایج به‌صورتی است که در شکل ۱ و شکل ۲ نمایش داده شده و در زیر آمده است:

```
Class 0: F1-Score: 0.9644, Recall: 0.9630, Precision: 0.9659
Class 1: F1-Score: 0.9710, Recall: 0.9720, Precision: 0.9701
Class 2: F1-Score: 0.9462, Recall: 0.9410, Precision: 0.9515
Class 3: F1-Score: 0.8941, Recall: 0.8950, Precision: 0.8932
Class 4: F1-Score: 0.9583, Recall: 0.9660, Precision: 0.9508
Class 5: F1-Score: 0.9151, Recall: 0.9220, Precision: 0.9084
Class 6: F1-Score: 0.9725, Recall: 0.9740, Precision: 0.9711
Class 7: F1-Score: 0.9688, Recall: 0.9640, Precision: 0.9737
Class 8: F1-Score: 0.9680, Recall: 0.9670, Precision: 0.9689
Class 9: F1-Score: 0.9688, Recall: 0.9630, Precision: 0.9747
Overall Accuracy: 0.9527
```



(ب) اتلاف (آ) دقت

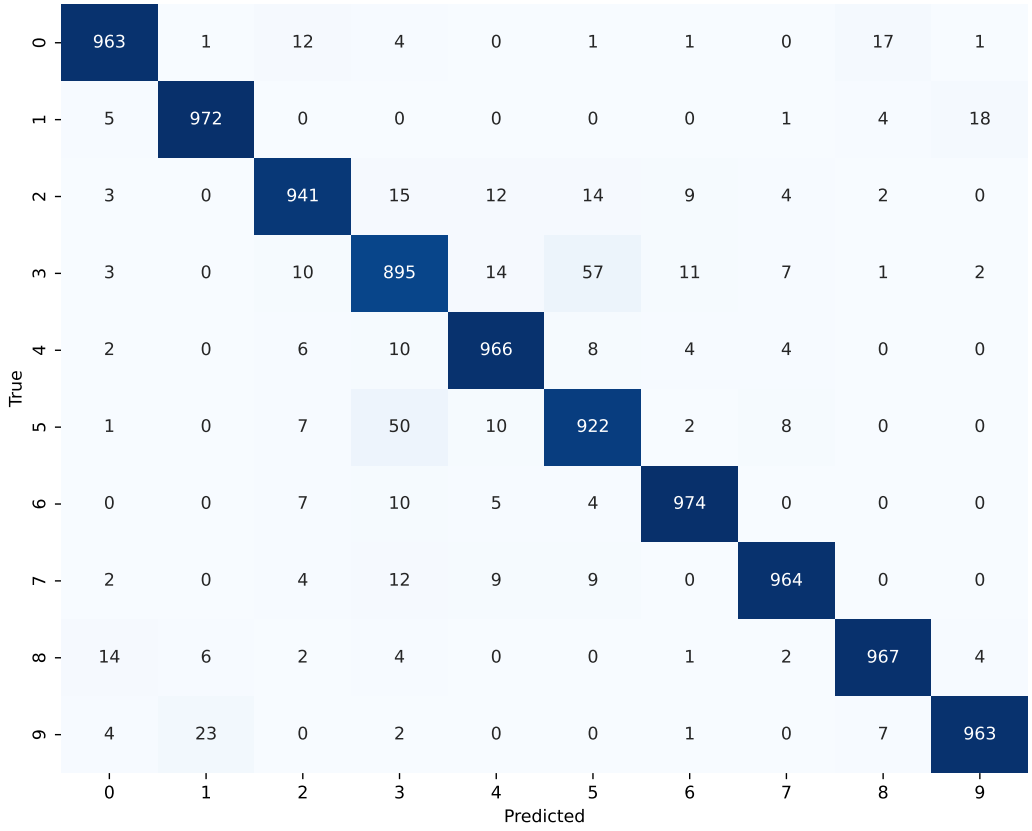
شکل ۱: نتایج نمودارهای دقت و اتلاف در VGG-19.

۳.۱ پاسخ قسمت ۲ - شبکهٔ ViT (تبدیل‌کنندهٔ تصویر)

برای هدف این سوال هم ابتدا تبدیل‌های مورد نیاز برای تصاویر را تعریف می‌کنیم تا فرآیند فراخوانی و پیش‌پردازش داده‌ها شفاف باشند و از فایل‌های ذخیره‌شده از قسمت قبل استفاده نکنیم. مطابق مقاله، تبدیل‌های مختلفی می‌توانند اعمال شوند، مانند تغییر اندازه (resize)، برگرداندن افقی (horizontal flip) و تبدیل تصویر به تنسور (convert to tensor). تمام این تبدیل‌ها در کلاس `transforms.Compose` به صورت زنجیره‌ای اعمال می‌شوند. در این حالت، تبدیلات عبارتند از: تغییر اندازهٔ تصویر به ابعاد 224*224 با استفاده از روش دوخطی (bilinear interpolation)، اعمال برگرداندن افقی برای افزایش تنوع داده‌ها.، و تبدیل تصویر به تنسور برای استفاده در مدل‌های پیچشی. در ادامه، مجموعه‌دادهٔ CIFAR-10 را دانلود و بارگیری می‌کنیم. دستهٔ آموزشی و آزمون را به‌صورت مجزا تعریف می‌کنیم. تبدیل‌های تعریف‌شده در مرحلهٔ قبلی با استفاده از پارامتر `transform` به هر دو مجموعه‌داده اعمال می‌شوند. مکان ذخیره‌سازی داده‌ها در پوشه data/. تعیین شده است. اگر داده‌ها قبلاً دانلود شده باشند، مجدداً دانلود نمی‌شوند. سپس، داده‌های آموزش و آزمون به عنوان دسته‌های مجزا تعریف می‌شوند. با استفاده از `torch.utils.data.DataLoader`، داده‌ها در دسته‌های مشخص (در این‌جا به تعداد ۶۴ تصویر در هر دسته) بارگیری و آمادهٔ استفاده در آموزش و ارزیابی مدل می‌شوند. داده‌های آموزش تصادفی مخلوط می‌شوند (`shuffle=True`)، اما داده‌های آزمون ترتیب خود را حفظ می‌کنند (`shuffle=False`).

در ادامه به دو صورت عمل می‌کنیم. یکی به‌صورت عادی و دیگری به‌صورتی که در صورت سوال ذکر و خواسته شده؛ یعنی تدقیق مدل با فقط تدقیق اوزان آخرین بلوک تبدیل‌کننده و MLP Head. این دستورات برای بارگیری مدل ViT-L۳۲ از پیش آموزش داده شده و سپس فریز کردن تمامی لایه‌ها به جز آخرین بلوک از ترانسفورمر و Head MLP استفاده می‌شوند.

در ابتدا دستوراتی می‌نویسیم که با استفاده از آن مدل ViT-L32 را با استفاده از کتابخانهٔ timm بارگیری کنیم. پارامتر `pretrained=True` نام مدل است که بر اساس اندازهٔ پچ‌ها و معماری ViT تعریف شده است. پارامتر `vit_large_patch32_224` هم نشان می‌دهد که مدل با وزن‌های پیش‌آموزش‌دیده بارگیری شود. و در نهایت، پارامتر `num_classes=10` تعداد کلاس‌های خروجی مدل را مشخص می‌کند که در اینجا برای مجموعه‌دادهٔ CIFAR-10، ۱۰ کلاس است. در ادامه دستوری نوشته‌ایم تا برای هر پارامتر در مدل، نام پارامتر و خود پارامتر را دریافت می‌کنیم. سپس، کاری می‌کنیم که در حالت عادی از بلوک گفته‌شده در مقاله و در حالت موردنظر سوال، آخرین بلوک ترنسفورمر و MLP Head غیرفریز شوند. در ادامه دستوراتی را برای تدقیق (Fine-



شکل ۲: نتایج ماتریس درهم‌ریختگی در VGG-19.

tuning) مدل و تعیین تابع هدف و بهینه‌ساز استفاده می‌نویسیم. در ابتدای امر، از کلاس `torch.nn.CrossEntropyLoss` برای محاسبه تابع خطا (loss) در مسألهٔ دسته‌بندی استفاده می‌کنیم. این تابع به صورت پیش‌فرض برای مسائل دسته‌بندی چند دسته‌ای مناسب است. سپس از کلاس `torch.optim.Adam` استفاده می‌کنیم و با استفاده از filter(lambda p: p.requires_grad, model.parameters())، فقط پارامترهایی که `requires_grad=True` دارند (یعنی پارامترهای آزاد برای به‌روزرسانی) را انتخاب می‌کنیم. در ادامه، دستگاه محاسباتی برای اجرای مدل را تعیین می‌کنیم. این دستور به صورتی نوشته شده که اگر دستگاه GPU (CUDA) در دسترس باشد، مدل را روی GPU اجرا می‌کند (device = torch.device("cuda"))، در غیر این صورت از پردازندهٔ مرکزی استفاده می‌کند (device = torch.device("cpu")). این کار با استفاده از `torch.cuda.is_available` انجام می‌شود. با انجام این مراحل، تابع هدف و بهینه‌ساز تعریف می‌شوند و مدل نیز به دستگاه محاسباتی منتقل می‌شود. این مراحل معمولاً قبل از شروع آموزش مدل به‌کار می‌روند. در ادامه، پارامترهای مربوط به آموزش را تعریف می‌کنیم که شامل تعداد دوره‌های آموزش (`num_epochs`)، ضریب کاهش نرخ یادگیری (`lr_factor`)، میزان تحمل (`lr_patience`)، و حداقل نرخ یادگیری (`min_lr`) می‌شود. هم‌چنین لیست‌هایی را برای ذخیرهٔ معیارهای آموزش و ارزیابی مدل تعریف می‌کنیم که شامل لیست خطاهای آموزش (`train_losses`)، دقت آموزش (`train_accs`)، خطاهای ارزیابی (`val_losses`) و دقت ارزیابی

(val_accs) است.

در انتها از کلاس torch.optim.lr_scheduler.ReduceLROnPlateau استفاده می‌کنیم و تمام تنظیمات مدنظر را اِعمال می‌کنیم. با انجام این فرآیند، حلقهٔ دوره‌های آموزش را تعریف می‌کنیم و با استفاده از range(num_epochs)، دوره‌های آموزش را ایجاد می‌کنیم. در هر دوره، مدل در حال آموزش قرار می‌گیرد (model.train) و خطا و دقت آموزش برای داده‌های آموزش محاسبه می‌شود. سپس مدل در حال ارزیابی قرار می‌گیرد (model.eval) و خطا و دقت ارزیابی برای داده‌های ارزیابی محاسبه می‌شود. سپس معیارهای محاسبه شده به لیست‌های مربوطه اضافه می‌شوند. در انتها نرخ یادگیری به عنوان یک پارامتر در بهینه‌ساز به‌روزرسانی می‌شود (lr_scheduler.step(val_acc)) و مقدار فعلی آن در هر چرخه نمایش داده می‌شود. در بخش بررسی بهبود عملکرد، عملکرد مدل با مدل بهتر قبلی مقایسه می‌شود. اگر دقت ارزیابی بهتر باشد، مدل بهتر به‌روزرسانی می‌شود و تعداد دوره‌های بدون بهبود صفر می‌شود. اگر دقت ارزیابی بهبود نیابد، تعداد دوره‌های بدون بهبود افزایش می‌یابد و اگر این تعداد به حد صبر تعیینی برسد، آموزش متوقف می‌شود. با انجام این مراحل، مدل به مدت تعداد دوره‌های آموزش مشخص شده آموزش می‌بیند و در هر دوره، خطا و دقت آموزش و ارزیابی ثبت می‌شود. همچنین، نرخ یادگیری به‌روزرسانی می‌شود و به‌روزرسانی بهبود عملکرد نیز انجام می‌شود. پس از انجام این کارها دستوراتی را می‌نویسیم تا نمودارها و معیارهای مختلفی را برای ارزیابی عملکرد مدل نمایش دهیم. دستورات به صورتی است که در برنامهٔ ۲ آورده شده است.

Program 2: ViT-L32 Implementation

```python
import torch
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import numpy as np
from sklearn.metrics import f1_score, recall_score, accuracy_score, precision_score
import timm

# Define the transformations
upscale_transform = transforms.Compose([
    transforms.Resize((224, 224), interpolation=transforms.InterpolationMode.BILINEAR),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor()  # Convert images to tensors
])

# Download and load the CIFAR-10 dataset
train_dataset = datasets.CIFAR10(root='./data', train=True, transform=upscale_transform, download
    =True)
test_dataset = datasets.CIFAR10(root='./data', train=False, transform=upscale_transform, download
    =True)

# Define the dataloaders
batch_size = 64
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

```
26
27 # Load the pre-trained ViT-L32 model
28 model = timm.create_model('vit_large_patch32_224', pretrained=True, num_classes=10)
29
30 # Freeze all layers except the last block of the transformer and the MLP head
31 for name, param in model.named_parameters():
32     if not name.startswith('Transformer/encoderblock_23') and not name.startswith('head'):
33         param.requires_grad = False
34
35 # Fine-tune the model
36 criterion = torch.nn.CrossEntropyLoss()
37 learning_rate = 0.0001
38 optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=
       learning_rate)
39 lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, factor=0.6, patience=1,
       min_lr=1e-7)
40
41 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
42 model.to(device)
43
44 num_epochs = 20
45 stop_after_epochs = 2  # Stop training after 5 epochs without improvement
46 best_val_acc = 0.0
47 epochs_without_improvement = 0
48
49 train_losses = []
50 train_accs = []
51 val_losses = []
52 val_accs = []
53
54 for epoch in range(num_epochs):
55     model.train()
56     train_loss = 0.0
57     train_acc = 0.0
58
59     for images, labels in train_loader:
60         images = images.to(device)
61         labels = labels.to(device)
62
63         optimizer.zero_grad()
64         outputs = model(images)
65         loss = criterion(outputs, labels)
66         loss.backward()
67         optimizer.step()
68
```

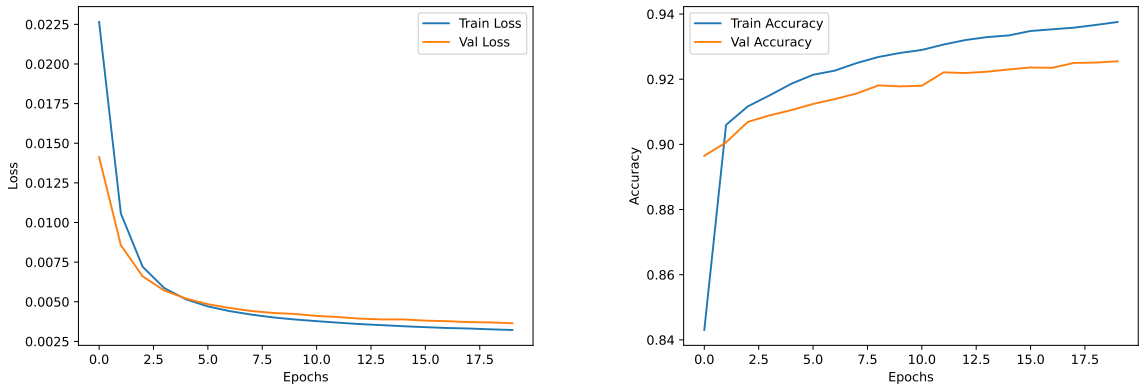
```
        train_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        train_acc += (predicted == labels).sum().item()

    train_loss /= len(train_loader.dataset)
    train_acc /= len(train_loader.dataset)

    train_losses.append(train_loss)
    train_accs.append(train_acc)

    model.eval()
    val_loss = 0.0
    val_acc = 0.0
    y_true = []
    y_pred = []

    with torch.no_grad():
        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)

            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            val_acc += (predicted == labels).sum().item()

            y_true.extend(labels.tolist())
            y_pred.extend(predicted.tolist())

    val_loss /= len(test_loader.dataset)
    val_acc /= len(test_loader.dataset)

    val_losses.append(val_loss)
    val_accs.append(val_acc)

    print(f"Epoch {epoch + 1}/{num_epochs}: Train Loss: {train_loss:.4f}, Train Acc: {train_acc
    :.4f}, "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")

    # Check if validation accuracy has improved
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        epochs_without_improvement = 0
    else:
```

```
        epochs_without_improvement += 1

    # Reduce learning rate if no improvement for a certain number of epochs
    if epochs_without_improvement >= stop_after_epochs:
        lr_scheduler.step(val_loss)
        if optimizer.param_groups[0]['lr'] < 1e-7:
            print("Training stopped as learning rate reached the minimum value.")
            break

# Plot val/train accuracy and loss
plt.figure()
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Val Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.savefig('lossplot33.pdf')
plt.show()

plt.figure()
plt.plot(train_accs, label='Train Accuracy')
plt.plot(val_accs, label='Val Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.savefig('accuracyplot33.pdf')
plt.show()

# Calculate and plot confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues', cbar=False)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.savefig('confusionmatrix33.pdf')
plt.show()

# Print F1-score, recall, accuracy, and precision for all classes
f1_scores = f1_score(y_true, y_pred, average=None)
recall_scores = recall_score(y_true, y_pred, average=None)
accuracy = accuracy_score(y_true, y_pred)
precision_scores = precision_score(y_true, y_pred, average=None)

for i in range(len(f1_scores)):
    print(f"Class {i}: F1-Score: {f1_scores[i]:.4f}, Recall: {recall_scores[i]:.4f}, "
```

```
            f"Precision: {precision_scores[i]:.4f}")

print(f"Overall Accuracy: {accuracy:.4f}")
```

نتایج به‌صورتی است که در شکل ۵ و شکل ۶ نمایش داده شده و در زیر آمده است:

```
Class 0: F1-Score: 0.9765, Recall: 0.9750, Precision: 0.9779
Class 1: F1-Score: 0.9756, Recall: 0.9800, Precision: 0.9713
Class 2: F1-Score: 0.9749, Recall: 0.9730, Precision: 0.9769
Class 3: F1-Score: 0.9258, Recall: 0.9360, Precision: 0.9159
Class 4: F1-Score: 0.9694, Recall: 0.9670, Precision: 0.9719
Class 5: F1-Score: 0.9285, Recall: 0.9410, Precision: 0.9163
Class 6: F1-Score: 0.9807, Recall: 0.9680, Precision: 0.9938
Class 7: F1-Score: 0.9828, Recall: 0.9730, Precision: 0.9929
Class 8: F1-Score: 0.9821, Recall: 0.9860, Precision: 0.9782
Class 9: F1-Score: 0.9729, Recall: 0.9690, Precision: 0.9768
Overall Accuracy: 0.9668
```

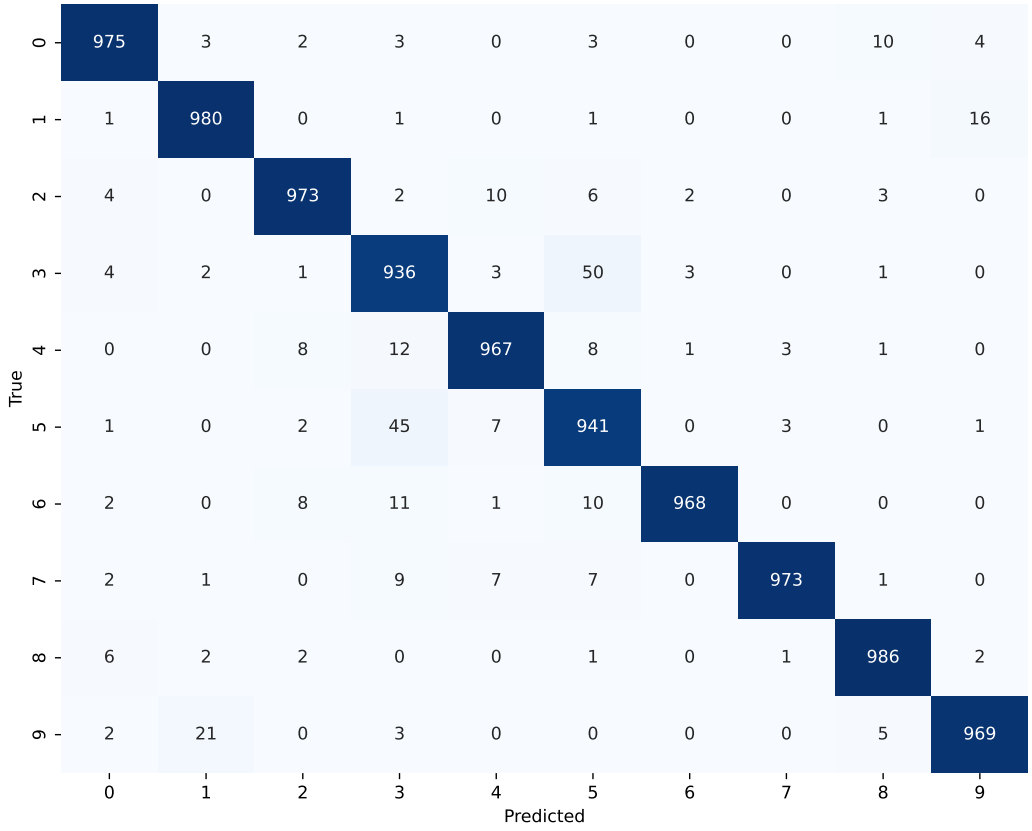


(آ) دقت (ب) اتلاف

شکل ۳: نتایج نمودارهای دقت و اتلاف در ViT-L32.

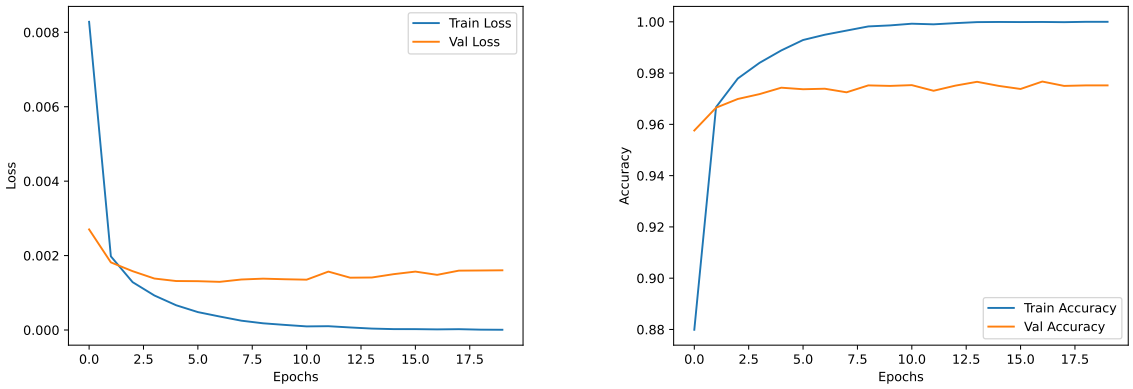
حال همین کار را برای مدل CaiT-S24 تکرار می‌کنیم و نتایج به‌صورتی است که در شکل ۵ و شکل ۶ نمایش داده شده و در زیر آمده است:

```
Class 0: F1-Score: 0.9755, Recall: 0.9760, Precision: 0.9750
Class 1: F1-Score: 0.9850, Recall: 0.9880, Precision: 0.9821
Class 2: F1-Score: 0.9744, Recall: 0.9700, Precision: 0.9788
Class 3: F1-Score: 0.9474, Recall: 0.9460, Precision: 0.9488
Class 4: F1-Score: 0.9735, Recall: 0.9730, Precision: 0.9740
Class 5: F1-Score: 0.9616, Recall: 0.9640, Precision: 0.9592
Class 6: F1-Score: 0.9870, Recall: 0.9850, Precision: 0.9890
Class 7: F1-Score: 0.9780, Recall: 0.9790, Precision: 0.9770
Class 8: F1-Score: 0.9880, Recall: 0.9900, Precision: 0.9861
```



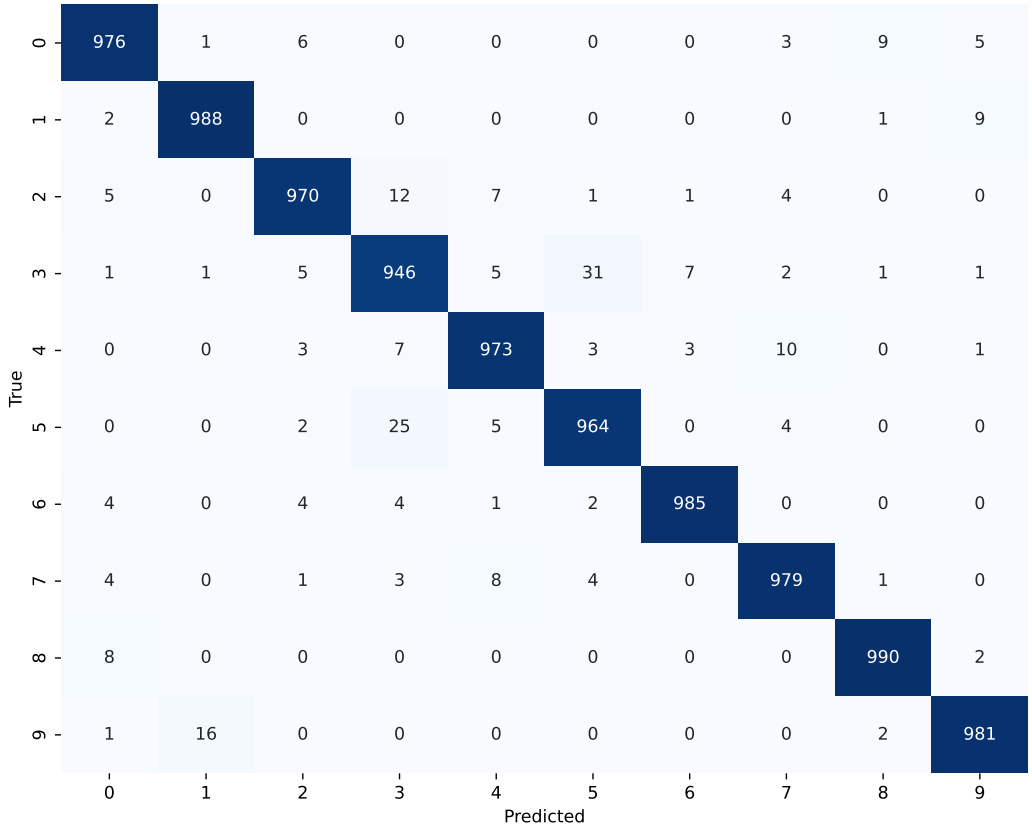
شکل ۴: نتایج ماتریس درهم‌ریختگی در ViT-L32.

```
Class 9: F1-Score: 0.9815, Recall: 0.9810, Precision: 0.9820
Overall Accuracy: 0.9752
```



(ب) اتلاف (آ) دقت

شکل ۵: نتایج نمودارهای دقت و اتلاف در CaiT-S24.



شکل ۶: نتایج ماتریس درهم‌ریختگی در CaiT-S24.