

به نام خدا

گزارش پیاده سازی مقاله

Investigating Transfer Learning Capabilities of Vision Transformers and CNNs by Fine-Tuning a Single Trainable Block

نویسندگان: Durvesh Malpure, Onkar Litake, Rajesh Ingle

پیاده سازی و اجرا: محمدمهدی برقی



دانشگاه تهران

دانشکده گان فنی

خرداد ماه ۱۴۰۳

فهرست

۳.....	به کارگیری مدل‌های ترنسفرمر ۱۰ در طبقه‌بندی تصاویر
۳.....	۲.۱. آشنایی با ترنسفورمرهای تصویر
۵.....	۲.۲. لود و پیش‌پردازش دیتاست
۷.....	۲.۳. Fine-Tuning شبکه کانولوشنی
۱۵.....	۲.۴. Fine-Tune شبکه ترنسفرمر
۲۲.....	۲.۴.۲. مدل ترنسفورمری دیگری
۲۷.....	۲.۵. مقایسه نتایج

به کارگیری مدل‌های ترنسفرمر ۱۰ در طبقه‌بندی تصاویر

۲.۱. آشنایی با ترنسفرمرهای تصویر

الف) شبکه vision transformer (ViT) همانگونه که از نامش پیداست ساختاری هست که به منظور استفاده در پردازش تصویر بر مبنای Transformer ها طراحی شده اند. و به گونه ای جدید از transformer ها که اکثرا برای وظایف، پردازش زبان طبیعی مورد استفاده قرار می‌گیرند برای تشخیص تصاویر به جای استفاده از CNN ها استفاده شده اند.

کارکرد این شبکه تا جایی که من از مقاله AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE به هر تصویر به عنوان یک کلمه نگاه می‌کند، به طور تخصصی تر هر تصویر با ابعاد $H*W*C$ به مجموعه ای از batch های دوبعدی با ابعاد $x*x$ که در این مقاله $X=16$ است تقسیم می‌شوند. و سپس هر بردار دو بعدی به یک بردار تخت یک بعدی تبدیل می‌شود و سپس به صورت توالی ای از توکن ها وارد مدل transformer ای می‌شود و ای باعث می‌شود تا رابطه اجزای مختلف تصویر حفظ شده و ضمن ویژگی هایی که در CNN داشتیم، بتوانیم توجهی به نقاط خاصی از تصویر را نیز افزایش یا کاهش دهیم.

لازم به ذکر است که نکته حائز اهمیتی که برای من خیلی مهم بود در نظر گرفتن یک embedding موقعیتی برای حفظ اهمیت موقعیت اجزای یک تصویر است.

در رابطه با عملکرد نیز به نظر میرسد به دلیل افزوده شدن ویژگی های ذکر شده نتایج بسیار خوبی نیز به همراه داشته است. و گاهی عملکرد بهینه تر و دقیق تر نسبت به CNN ها که برای این امر طراحی شده اند داشته اند..

ب) در رابطه با بخش های مختلف معماری ViT می‌توان به این موارد اشاره کرد:

۱. پیش پردازش تصویر و تبدیل به batch ها کوچکتر و در ادامه تبدیل به تنسور و بردار دو بعدی و در نهایت یک بردار یک بعدی تخت

۲. موقعیت دهی به batch و در نظر گرفتن موقعیت هر batch در تصویر با استفاده از embedding موقعیت ها در تصویر

۳. token class یک لایه یادگیرنده ترسرفری که برای طبقه بندی اولیه مورد استفاده قرار می‌گیرد.

۴. transformer encoder که توالی توکن ها در این لایه به انکودر ترنسفرمر و وارد می شود و فرایند غیرخطی سازی بر روی توکن ها صورت می گیرد

۵. لایه خروجی: در نهایت خروجی یک کلاس از تصاویر است که به تصویر یک طبقه بندی تخصیص داده ها است و معمولا در fine-tune این لایه unfreeze می شود تا با توجه به تعداد کلاس ها آموزش صورت بپذیرد.

ج) به نظر من از مهم ترین ایراداتی که به VIT وارد است، نیازمندی آن به مقادیر زیادی از داده ها است که با استفاده از شبکه های ترنسفرمری از پیش آموزش دیده می توان تا حدودی اثر این مشکل را کمتر کرد. ایراد دومی که در مقاله به این اشاره شده است، عدم امکان ایجاد همبستگی بین ویژگی ها (feature) ها در تصویر است.

و ایراد سوم هم پیش پردازش پیچیده و طولانی مدتی است که این مدل ها دارند و مدت زمان یادگیری و ارزیابی را به شدت افزایش می دهند و برای تسک های real-time مناسب نیستند.

۲.۲. لود و پیش پردازش دیتاست

برای شروع در ابتدا گوگل کولب را برای ذخیره سازی مدل های Fine-Tune شده به گوگل داریو متصل کردم:

Q2 - Initialize

Install and Import libraries

```
[ ] from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

در مرحله بعدی سه کتابخانه که در این تمرین به آن ها نیاز داریم، اما به صورت دیفالت بر روی گوگل کولب نصب نیستند را نصب می کنیم:

```
!pip install timm
!pip install datasets
!pip install transformers

Collecting timm
  Downloading timm-1.0.3-py3-none-any.whl (2.3 MB)
    2.3 MB 2.3 MB/s eta 0:00:00
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (from timm) (2.3.0+cu121)
Requirement already satisfied: torchvision in /usr/local/lib/python3.10/dist-packages (from timm) (0.18.0+cu121)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.10/dist-packages (from timm) (6.0.1)
Requirement already satisfied: huggingface_hub in /usr/local/lib/python3.10/dist-packages (from timm) (0.23.2)
Requirement already satisfied: safetensors in /usr/local/lib/python3.10/dist-packages (from timm) (0.4.3)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from huggingface_hub->timm) (3.14.0)
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.10/dist-packages (from huggingface_hub->timm) (2023.6.0)
Requirement already satisfied: packaging>=20.9 in /usr/local/lib/python3.10/dist-packages (from huggingface_hub->timm) (24.0)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from huggingface_hub->timm) (2.31.0)
Requirement already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.10/dist-packages (from huggingface_hub->timm) (4.66.4)
Requirement already satisfied: typing_extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-packages (from huggingface_hub->timm) (4.12.1)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch->timm) (1.12.1)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch->timm) (3.3)
```

در ادامه تمامی کتابخانه های مورد نیازمان را import می کنیم:

```
[ ] import os
import time
import math
import random
import datetime
import numpy as np
import pandas as pd
import seaborn as sns
from scipy import stats
import matplotlib as plt
import matplotlib.pyplot as plt

# =====
import torch
import timm
import torchvision
import torchvision.transforms as transforms
from timm.models.vision_transformer import VisionTransformer
import torch.nn as nn
from torchvision import models
from torch.utils.data import DataLoader
```

و از آنجایی که مراحل بعدی می خواهیم پردازش را بر روی GPU انجام دهیم در همین ابتدا مقدار متغیر device را تعیین می کنیم تا در ادامه قبل از آموزش مدل ها آن ها را به device منتقل کنیم:

```
[ ] device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"device:{device}")
```

device:cuda:0

در این مرحله به سراغ لود و پیش پردازش دیتاست می رویم:

در ابتدا با استفاده از transforms تغییراتی و پیش پردازش هایی که انتظار داریم در ابتدا بر روی تصاویر انجام دهیم را مشخص می کنیم:

load dataset

```
[ ] transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

که در این قسمت من ابتدا تصویر را طبق اطلاعات مقاله به 224×224 تغییر ابعاد می دهیم، تصویر را به یک tensor تبدیل می کنیم و هر یک از ابعاد تصویر را نرمالایز می کنم.

سپس طبق کد زیر دیتاست CIFAR10 را طبق مقاله و خواسته سوال در ابتدا در دو دسته Train_set و Test_Set با کمک کتابخانه torchvision.datasets دریافت کردم:

```
batch_size = 32
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                       download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

و از آنجایی که مشکل RAM برای مدل های مختلف برای من ایجاد نشد و نیاز به سرعت بیشتری برای Fine-Tune شدن مدل هایم داشتم، اندازه Batch_size را ۳۲ تایی در نظر گرفتم.

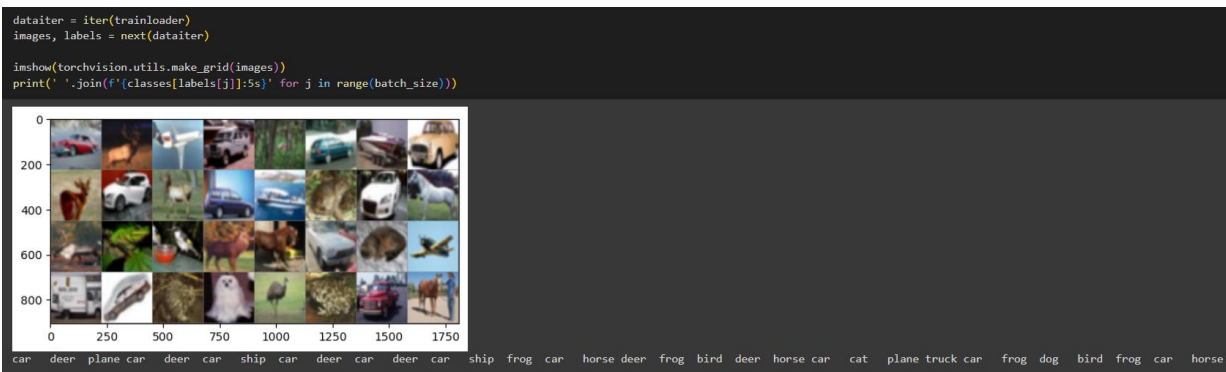
همچنین با توجه به توضیحاتی که برای این دیتاست وجود دارد یک مجموعه از ۱۰ کلاس موجود در این دیتاست را نیز ایجاد کردم.

برای بررسی بیشتر دیتاست نیز یک تابع بدین صورت تعریف کردم:

check loaded dataset

```
[ ] def imshow(img):
    img = img / 2 + 0.5
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

که نتیجه آن برای دیتاستی که لود کردم بدین صورت است:



۲.۳ Fine-Tuning شبکه کانولوشنی

الف) برای انتخاب شبکه کانولوشنی:

Model	Model type	Trainable Parameters	Validation Accuracy(%)	Pretrained on
Resnet152V2[9]	CNN	15,497,994	83.583	ImageNet1K
EfficientNetV2L[21]	CNN	7,020,960	90.130	ImageNet1K
VGG-19[17]	CNN	9,573,130	92.784	ImageNet1K
NFNetF6[3]	CNN	12,012,043	94.350	ImageNet1K
Densenet201[10]	CNN	6,982,400	94.757	ImageNet1K
ViT-L32[7]	Transformer	12,863,242	95.467	ImageNet21K
Swin-B224[15]	Transformer	12,868,650	93.580	ImageNet1K
CoaT-LiteSmall[25]	Transformer	3,308,298	94.100	ImageNet1K
CaiTS24[26]	Transformer	1,877,130	96.000	ImageNet1K
DeiTBaseDistilled[22]	Transformer	7,488,276	96.450	ImageNet1K

Table 1. Comparison of accuracy obtained on CIFAR-10 validation set by all the models along with their respective trainable parameter and the dataset it was pretrained upon.

با توجه به جدول ۱ مقاله من مدل densenet201 که هم کمترین پارامتر و هم بیشترین دقت را در بین مدل‌های CNN دارد را انتخاب کردم و در ادامه ابتدا بدین صورت این مدل را لود کردم:



این مدل نیز به صورت دیفالت با دیتاست imagenet1k آموزش دیده است و برای استفاده از مدل pre-train شده آن، کافی است تا مقدار پارامتر pretrained را برای آن true قرار دهیم.

در مرحله بعد طبق خواسته مقاله، می بایست تمامی لایه ها مگر موارد گفته شده در مقاله را قبل از آموزش مجدد با دیتاست، Freeze کنیم، که طبق مقاله برای مدل densenet201 این لایه ها unfreeze شده اند:

- VGG-19: block5 conv1
- ResNet152V2: conv5 block1 preact bn
- DenseNet201: conv5 block1 0. bn
- EfficientNetV2-L: Last InvertedResidual of last Sequential block
- NFNetF6: Last NormFreeBlock of last Sequential block
- CoaT-LiteSmall: Last SerialBlock of the serial.blocks 4
- CaiT-S24: Last block of the blocks token only module
- DeiTBaseDistilled: 12th Transformer Block

به همین سبب من ابتدا تمامی لایه ها مدل از پیش آموزش دیده را فریز می کنم:

▼ freeze and unfreeze layers

```
[ ] for param in model.parameters():  
    param.requires_grad = False
```

سپس همانطور که توضیح داده شد لایه های مورد نظر را unfreeze می کنیم:

```
[ ] for name, param in model.named_parameters():  
    if "denseblock4.denselayer16" in name:  
        param.requires_grad = True
```

ب) در ادامه با توجه به لایه ها و پارامترهایی که unfreeze شدند و می توانند آموزش ببینند، می توانیم تعداد کل پارامترهایی که می توانند آموزش ببینند را اینگونه بررسی کنیم:

▼ B - trainable parameters

```
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)  
print(f'Number of trainable parameters: {trainable_params}')
```

Number of trainable parameters: 235210

که همانطور که مشخص شده است در این مدل ۲۳۵۲۱۰ پارامتر قابل آموزش دیدن وجود دارد.

ج) حال پس از تعیین پارامترهایی که قابلیت آموزش دیدن را دارند و آماده سازی اولیه مدل می‌بایست آن را با داده های CFAR10، Fine-Tune کنیم به همین سبب ابتدا برخی از HyperParametra های مدل را طبق آنچه که در مقاله ذکر شده است را تعیین می کنیم:

▼ C - fine-tune

```
[ ] criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=0.0001)
```

که مهم ترین آن ها می توان به تعیین Adam به عنوان optimizer و همچنین تعیین نرخ یادگیری ۰.۰۰۰۱ برای آموزش مدل.

سپس دو تابع آموزش و ارزیابی مدل را تعریف کردم که بدین صورت هستند:

تابع آموزش:

▼ train function

```
[ ] def train(model, trainloader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for inputs, labels in trainloader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * inputs.size(0)
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

    epoch_loss = running_loss / len(trainloader.dataset)
    epoch_accuracy = 100. * correct / total

    return epoch_loss, epoch_accuracy
```

که در آن همانطور که قابل مشاهده هست مدل تعیین شده به صورت دائم به آن برچسب ها و داده های تنسور تصاویر داده می شود با استفاده از تابع اپتیمایز در هر گام برای کاهش loss، backward prediction انجام می گیرد.

برای تابع validation نیز داریم:

validation function

```
def validate(model, testloader, criterion, device):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in testloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            running_loss += loss.item() * inputs.size(0)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

    epoch_loss = running_loss / len(testloader.dataset)
    epoch_accuracy = 100. * correct / total

    return epoch_loss, epoch_accuracy
```

که این بار ارزیابی مدل بر اساس مجموعه داده های تست تعیین شده به مدل داده خواهد شد و دقت آن به ازای predicted هایی که داشته است، محاسبه می گردند.

پس تعریف این دو تابع اصلی برای مدل، ابتدا همانطور که قبلا هم اشاره کردیم، مدل را برای سرعت بیشتر به GPU انتقال می دهیم:

training loop

```
model.to(device)
```

سپس تعداد ایپاک و ۶ لیست برای ذخیره سازی هر یک از مقادیر دقت مدل بر روی داده های آموزش و ارزیابی، میزان loss برای داده های آموزشی و ارزیابی و زمان های مورد نیاز برای آموزش و ارزیابی در نظر می گیریم که در نهایت بتوانیم بر روی این داده ها تحلیل کرده و مدل ها را با یکدیگر مقایسه کنیم، همچنین شایان به ذکر است که با توجه به طولانی بودن زمان fine-tune شدن مدل و همچنین محدودیت زمانی بسیار بالای google colab برای استفاده از GPU بر خلاف مقاله (مقاله ۲۰ ایپاک برای آموزش هر مدل زمان گذاشته است.)، ما ۱۰ ایپاک برای تنظیم مجدد پارامترها در نظر گرفتیم:

```
[ ] num_epochs = 10
    train_losses, train_accuacies = [], []
    val_losses, val_accuacies = [], []
    train_times, val_times = [], []
```

در ادامه وارد حلقه آموزش شده و به ازای هر ایپاک آموزش و ارزیابی مدل را بدین صورت انجام دادیم و در هر مرحله داده های مورد نیاز برای ارزیابی مدل در لیست های مربوط ذخیره سازی شدند:

```

for epoch in range(num_epochs):
    start_time = time.time()
    train_loss, train_accuracy = train(model, trainloader, criterion, optimizer, device)
    end_time = time.time()
    train_times.append(end_time - start_time)

    start_time = time.time()
    val_loss, val_accuracy = validate(model, testloader, criterion, device)
    end_time = time.time()
    val_times.append(end_time - start_time)

    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)
    val_losses.append(val_loss)
    val_accuracies.append(val_accuracy)

    print(f"\n\n\n----- Epoch {epoch+1}/{num_epochs} -----")
    print(f"Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.2f}%\n "
          f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_accuracy:.2f}%\n "
          f"Train Time: {train_times[-1]:.2f}s, Validation Time: {val_times[-1]:.2f}s\n")

```

که نتیجه هر ایپاک بدین صورت بود:

-----Epoch 1/10-----

Train Loss: 0.9497, Train Accuracy: 76.65%

Validation Loss: 0.4877, Validation Accuracy: 87.06%

Train Time: 294.05s, Validation Time: 53.23s

-----Epoch 2/10-----

Train Loss: 0.4435, Train Accuracy: 87.52%

Validation Loss: 0.3472, Validation Accuracy: 89.50%

Train Time: 299.63s, Validation Time: 52.47s

-----Epoch 3/10-----

Train Loss: 0.3323, Train Accuracy: 90.38%

Validation Loss: 0.3048, Validation Accuracy: 90.37%

Train Time: 300.31s, Validation Time: 53.05s

-----Epoch 4/10-----

Train Loss: 0.2661, Train Accuracy: 92.29%
Validation Loss: 0.2733, Validation Accuracy: 91.28%
Train Time: 300.30s, Validation Time: 53.34s

-----Epoch 5/10-----

Train Loss: 0.2212, Train Accuracy: 93.65%
Validation Loss: 0.2662, Validation Accuracy: 91.14%
Train Time: 299.94s, Validation Time: 53.54s

-----Epoch 6/10-----

Train Loss: 0.1864, Train Accuracy: 94.75%
Validation Loss: 0.2560, Validation Accuracy: 91.68%
Train Time: 300.60s, Validation Time: 53.16s

-----Epoch 7/10-----

Train Loss: 0.1576, Train Accuracy: 95.72%
Validation Loss: 0.2500, Validation Accuracy: 91.76%
Train Time: 300.09s, Validation Time: 53.33s

-----Epoch 8/10-----

Train Loss: 0.1313, Train Accuracy: 96.51%
Validation Loss: 0.2565, Validation Accuracy: 91.41%
Train Time: 300.44s, Validation Time: 53.28s

-----Epoch 9/10-----

Train Loss: 0.1128, Train Accuracy: 97.09%
Validation Loss: 0.2506, Validation Accuracy: 91.70%
Train Time: 299.87s, Validation Time: 52.95s

-----Epoch 10/10-----

Train Loss: 0.0955, Train Accuracy: 97.72%

Validation Loss: 0.2605, Validation Accuracy: 91.44%

Train Time: 300.62s, Validation Time: 52.84s

در نهایت از آنجایی که مجموع آموزش این مدل ها بسیار زمان گیر بود من هر مدل را پس از آموزش ذخیره نیز کردم:

```
save model

model_save_path = '/content/drive/MyDrive/Deep_learning/HW5/fine_tuned_densenet201_10.pth'

torch.save({
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'train_losses': train_losses,
    'train_accuracies': train_accuracies,
    'val_losses': val_losses,
    'val_accuracies': val_accuracies
}, model_save_path)

print('Model saved!!')
```

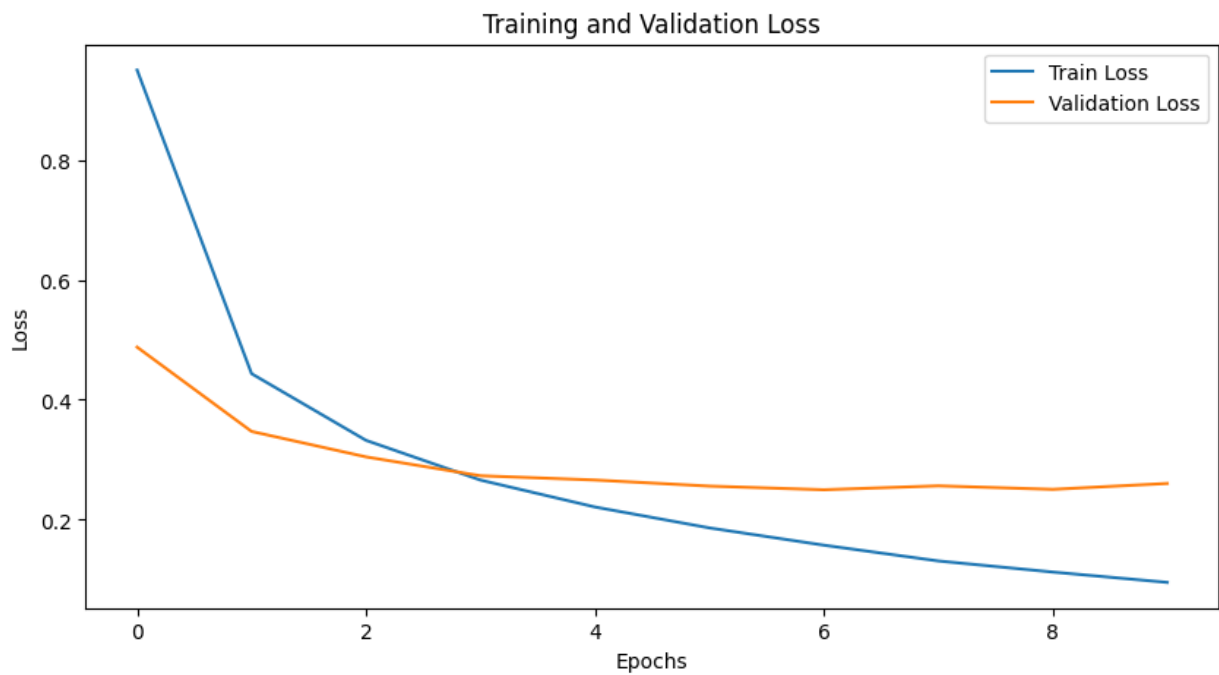
(د) برای رسم نمودار دقت و تابع هزینه بر روی داده های آموزشی و اعتبارسنجی داریم:
برای تابع هزینه داریم:

```
D - Plot the training and validation

loss

plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

که نتیجه آن بدین صورت است:

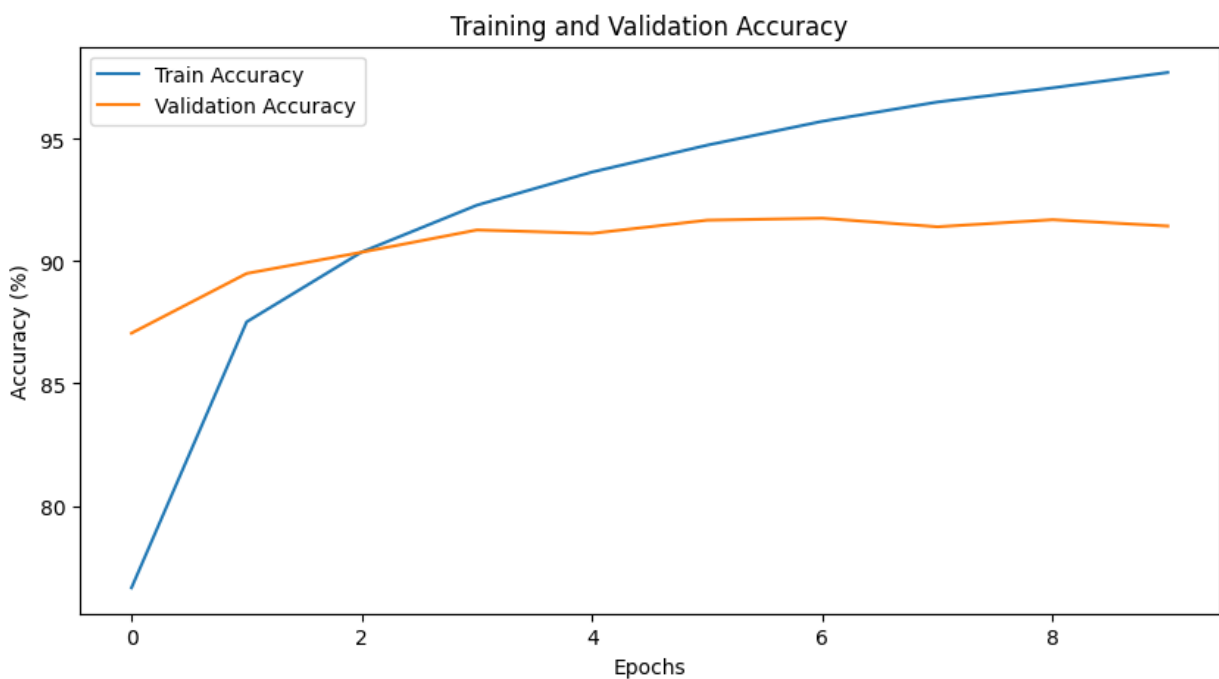


و برای دقت مدل نیز داریم:

accuracy

```
plt.figure(figsize=(10, 5))
plt.plot(train_accuracies, label='Train Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.legend()
plt.show()
```

که نتیجه آن بدین صورت است:



که طبق نتیجه دقت نهایی برروی داده های آموزشی ۹۷.۷۲٪ و برروی داده ها اعتبارسنجی ۹۱.۴۴٪ است. و مقدار تابع هزینه برای داده های آموزشی در این مدل برابر ۰.۰۹۵۵ برای داده های اعتبارسنجی: ۰.۲۶۰۵ است.

ه) در نهایت برای این مدل هم با توجه به زمان هایی که در حین آموزش و ارزیابی جمع آوری کردیم برای میانگین مدت زمان ها داریم:

```

    ▾ E - calculate training and validation time (bonus)

    [ ] total_train_time = sum(train_times)
        total_val_time = sum(val_times)
        average_train_time = total_train_time / num_epochs
        average_val_time = total_val_time / num_epochs

        print(f"Average training time per epoch: {average_train_time:.2f}s")
        print(f"Average validation time per epoch: {average_val_time:.2f}s")

    ⚙ Average training time per epoch: 299.59s
      Average validation time per epoch: 53.12s
  
```

۲.۴ Fine-Tune شبکه ترنسفرمر

برای شبکه ترنسفرمری DeiTBaseDistilled من با توجه به اینکه در مرحله آموزش این مدل به مشکل عدم برابر تعداد لایه های ورودی بر اساس داده ها برخوردیم با توجه به مدل PRE-TRAIN شده ابتدا این مدل را با کمک کتابخانه tinn به صورت از پیش آموزش دیده برروی داده های imagenet1k را لود کرده و سپس یک لایه به عنوان head به ابتدای این مدل اضافه کردم.

بدین صورت:

```

    [ ] class CustomDeiT(nn.Module):
        def __init__(self, num_classes):
            super(CustomDeiT, self).__init__()
            self.model = timm.create_model('deit_base_distilled_patch16_224', pretrained=True)
            self.model.head = nn.Linear(self.model.head.in_features, num_classes)
            self.model.head_dist = nn.Linear(self.model.head_dist.in_features, num_classes)

            def forward(self, x):
                x = self.model.forward_features(x)
                cls_token = self.model.head(x[:, 0])
                dist_token = self.model.head_dist(x[:, 1])
                return cls_token, dist_token
  
```

سپس با توجه به مقاله تمامی لایه ها را freeze کرده و لایه ۱۲ (لایه blocks.11 با توجه به اینکه zero-index است) را unfreeze کردم:

▼ freeze and unfreeze layers

```
[ ] for param in model.parameters():
    param.requires_grad = False

[ ] for name, param in model.named_parameters():
    if "blocks.11" in name:
        param.requires_grad = True

[ ] model.model.head = nn.Linear(model.model.head.in_features, num_classes)
    model.model.head_dist = nn.Linear(model.model.head_dist.in_features, num_classes)
```

ب) با آماده سازی این مدل و freeze کردن بسیاری از پارامترها به تعداد ۷۱۰۳۲۵۲ پارامتر برای آموزش باقی ماند که با کد زیر بررسی شد:

▼ B - trainable parameters

```
[ ] trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    print(f'Number of trainable parameters: {trainable_params}')
```

Number of trainable parameters: 7103252

ج) برای fine-tune کردن داده های CFAR10 بر روی این مدل نیز ابتدا مانند مدل قبل optimizer و learning rate را تعیین می کنیم:

▼ C - fine-tune

```
[ ] criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=0.0001)
```

سپس تابع های آموزش و اعتبارسنجی را با توجه به تغییراتی که روی مدل برای ایجاد تناسب بین لایه های ورودی انجام دادیم می افزاییم:

تابع آموزش:

▼ train function

```
[ ] def train(model, trainloader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for inputs, labels in trainloader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        cls_token, dist_token = model(inputs)
        loss = criterion(cls_token, labels) + criterion(dist_token, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * inputs.size(0)
        _, predicted = cls_token.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()
    epoch_loss = running_loss / len(trainloader.dataset)
    epoch_accuracy = 100. * correct / total
    return epoch_loss, epoch_accuracy
```


تابع اعتبارسنجی:

validation function

```
[ ] def validate(model, testloader, criterion, device):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testloader:
            inputs, labels = inputs.to(device), labels.to(device)
            cls_token, dist_token = model(inputs)
            loss = criterion(cls_token, labels) + criterion(dist_token, labels)
            running_loss += loss.item() * inputs.size(0)
            _, predicted = cls_token.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()
    epoch_loss = running_loss / len(testloader.dataset)
    epoch_accuracy = 100. * correct / total
    return epoch_loss, epoch_accuracy
```

حال برای آموزش و fine-tune کردن مدل، ابتدا مجدد آن را برای سرعت بیشتر به GPU انتقال می دهیم:

training loop

```
[ ] model.to(device)
```

Show hidden output

و تعداد ایپاک ها (مجددا ۱۰ مرتبه) و ۶ لیست مورد نیاز را تعیین و ایجاد می کنیم:

```
[ ] num_epochs = 10
    train_losses, train_accuracies = [], []
    val_losses, val_accuracies = [], []
    train_times, val_times = [], []
```

و مجددا مانند مدل قبلی به تعداد ایپاک ها فرایند آموزش، اعتبارسنجی و مقداردهی به پارامترها را تکرار می کنیم:

```
for epoch in range(num_epochs):
    start_time = time.time()
    train_loss, train_accuracy = train(model, trainloader, criterion, optimizer, device)
    end_time = time.time()
    train_times.append(end_time - start_time)

    start_time = time.time()
    val_loss, val_accuracy = validate(model, testloader, criterion, device)
    end_time = time.time()
    val_times.append(end_time - start_time)

    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)
    val_losses.append(val_loss)
    val_accuracies.append(val_accuracy)

    print(f"\n\n\n----- Epoch {epoch+1}/{num_epochs} -----")
    print(f"Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.2f}%\n "
          f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_accuracy:.2f}%\n "
          f"Train Time: {train_times[-1]:.2f}s, Validation Time: {val_times[-1]:.2f}s\n")
```

که نتیجه آن بدین صورت است:

-----Epoch 1/10-----

Train Loss: 0.3769, Train Accuracy: 93.92%

Validation Loss: 0.2242, Validation Accuracy: 96.26%

Train Time: 627.09s, Validation Time: 108.95s

-----Epoch 2/10-----

Train Loss: 0.1320, Train Accuracy: 97.87%

Validation Loss: 0.2144, Validation Accuracy: 96.60%

Train Time: 626.31s, Validation Time: 108.55s

-----Epoch 3/10-----

Train Loss: 0.0512, Train Accuracy: 99.25%

Validation Loss: 0.2587, Validation Accuracy: 96.09%

Train Time: 625.69s, Validation Time: 108.44s

-----Epoch 4/10-----

Train Loss: 0.0194, Train Accuracy: 99.73%

Validation Loss: 0.2680, Validation Accuracy: 96.59%

Train Time: 625.90s, Validation Time: 108.58s

-----Epoch 5/10-----

Train Loss: 0.0111, Train Accuracy: 99.82%

Validation Loss: 0.3366, Validation Accuracy: 96.05%

Train Time: 627.12s, Validation Time: 108.79s

-----Epoch 6/10-----

Train Loss: 0.0134, Train Accuracy: 99.79%

Validation Loss: 0.3418, Validation Accuracy: 96.29%

Train Time: 626.60s, Validation Time: 108.83s

-----Epoch 7/10-----

Train Loss: 0.0069, Train Accuracy: 99.87%

Validation Loss: 0.3216, Validation Accuracy: 96.65%

Train Time: 625.90s, Validation Time: 108.55s

-----Epoch 8/10-----

Train Loss: 0.0072, Train Accuracy: 99.89%

Validation Loss: 0.3631, Validation Accuracy: 96.57%

Train Time: 626.58s, Validation Time: 108.51s

-----Epoch 9/10-----

Train Loss: 0.0092, Train Accuracy: 99.84%

Validation Loss: 0.3538, Validation Accuracy: 96.63%

Train Time: 627.03s, Validation Time: 108.60s

-----Epoch 10/10-----

Train Loss: 0.0070, Train Accuracy: 99.87%

Validation Loss: 0.3890, Validation Accuracy: 96.44%

Train Time: 626.19s, Validation Time: 108.79s

این مدل را هم پس از آموزش مجدداً بدین صورت ذخیره کردم:

▼ save model

```
[ ] model_save_path = '/content/drive/MyDrive/Deep_learning/HW5/fine_tuned_deit_base_distilled_10.pth'

torch.save({
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'train_losses': train_losses,
    'train_accuaries': train_accuaries,
    'val_losses': val_losses,
    'val_accuaries': val_accuaries
}, model_save_path)

print(f'Model saved!!')
```

(د) برای بررسی نمودار توابع هزینه و دقت نیز برای داده های آموزشی و اعتبارسنجی داریم:
مقدار تابع هزینه (loss):

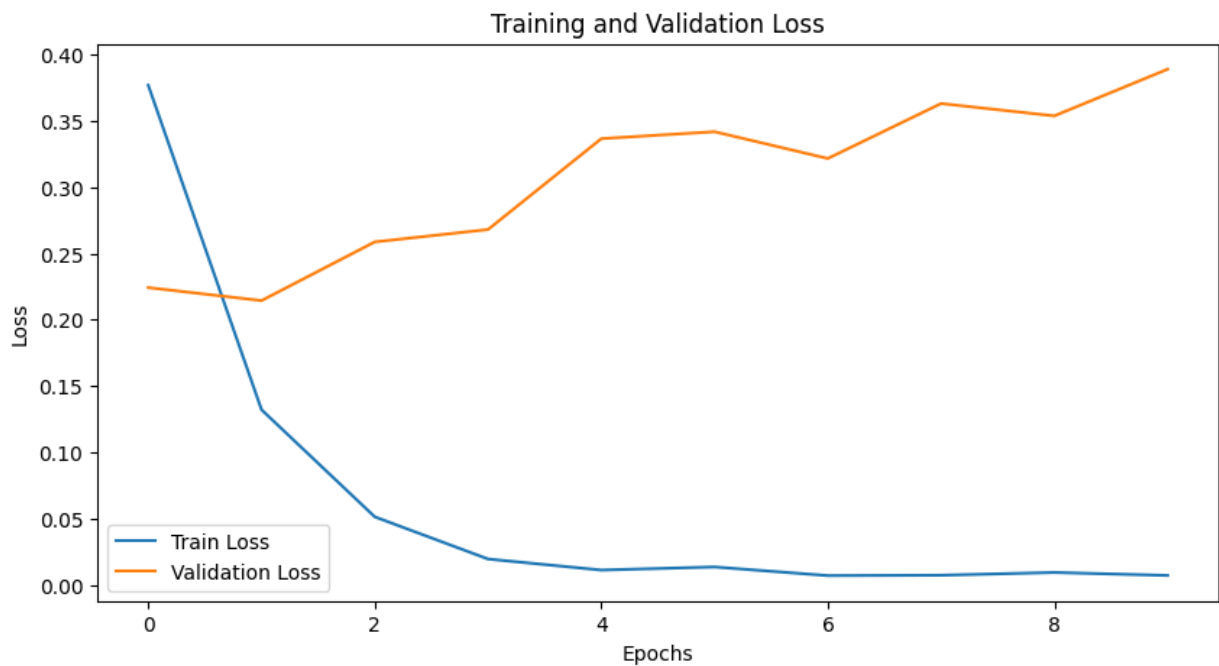
برای محاسبه تابع هزینه این کد را داریم:

▼ D - Plot the training and validation

▼ loss

```
[ ] plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Validation Loss')
plt.title("Training and Validation Loss")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

و نتیجه بدین صورت است:



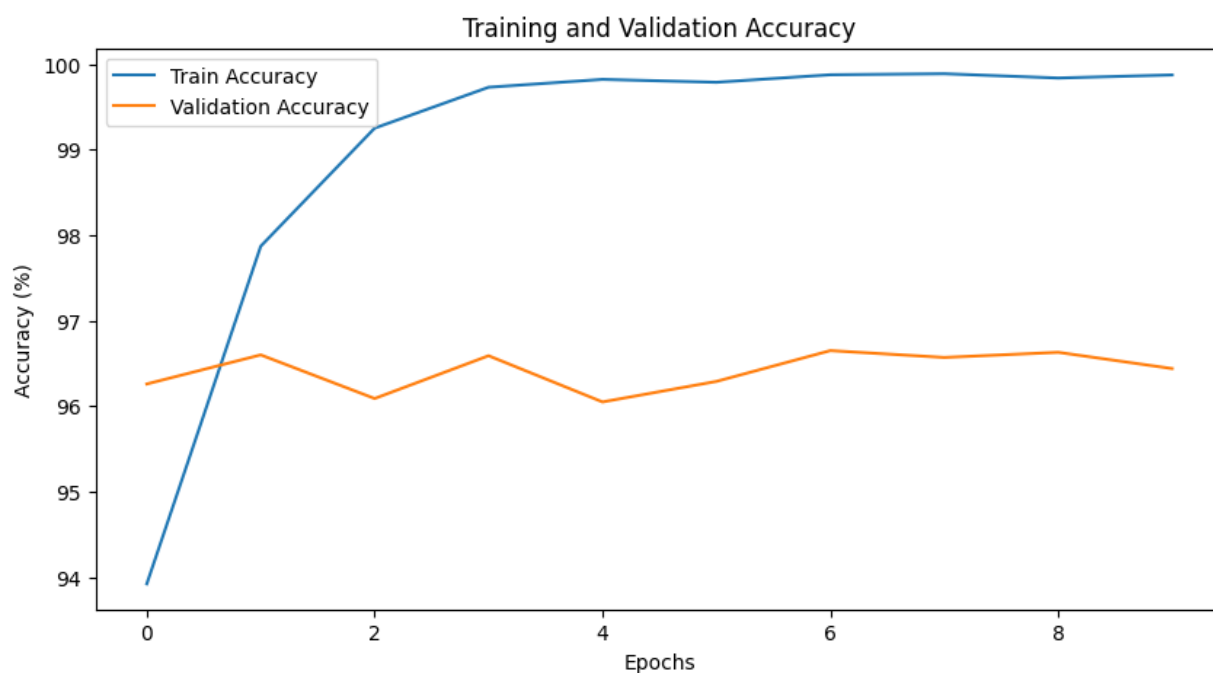
که نتیجه عجیبی است و مقدار تابع loss برای داده ها validation با افزایش ایپاک بر خلاف انتظار به جای کاهش در حال افزایش یافتن است و نشان می دهد که گویی با fine-tune کردن مدل داریم مقدار loss در حال افزایش است.

همچنین مقدار دقت نیز بدین صورت است:

```
▼ accuracy

plt.figure(figsize=(10, 5))
plt.plot(train_accuracies, label='Train Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.legend()
plt.show()
```

و نتیجه آن، این نمودار است:



که در نهایت نشان می دهد دقت نهایی مدل در داده های آموزشی برابر ۹۹.۸۷٪ بوده و برای داده های اعتبارسنجی این دقت برابر ۹۶.۴۴٪ است.

و مقدار تابع هزینه نیز در داده های آموزشی برابر ۰.۰۰۷ بوده و در داده های اعتبارسنجی ۰.۳۸۹۰ است.

ه) برای محاسبه مدت زمان آموزش و اعتبارسنجی به ازای هر ایپاک نیز داریم:

▼ E - calculate training and validation time (bonus)

```
total_train_time = sum(train_times)
total_val_time = sum(val_times)
average_train_time = total_train_time / num_epochs
average_val_time = total_val_time / num_epochs

print(f"Average training time per epoch: {average_train_time:.2f}s")
print(f"Average validation time per epoch: {average_val_time:.2f}s")
```

Average training time per epoch: 626.44s
Average validation time per epoch: 108.66s


که نشان می دهد به طور متوسط برای گام آموزش مدل در ایپاک ۶۲۸.۴۴ ثانیه زمان صرف شده (بیشتر از دو برابر مدل کانولوشنی که ۲۹۹.۵۹ ثانیه زمان نیاز دارد.) و برای اعتبارسنجی مدل زمان ۱۰۸.۶۶ ثانیه ثبت شده است که این زمان اعتبارسنجی نسبت به ۵۳.۱۲ ثانیه مورد نیاز در مدل کانولوشنی، مجددا طولانی تر است.

۲.۴.۲. مدل ترنسفورمری دیگری

از آنجایی که نتایج بدست آمده در مدل DeiTBaseDistilled عجیب بود من fine-tune کردن را بروی مدل CaiTS24 که تعداد پارامتر آموزشی کمتری دارد نیز امتحان و اجرا کردم که نتایج و کدهای آن بدین صورت است:

▼ Q2 - another transformer model

```
[ ] model = timm.create_model('cait_s24_224', pretrained=True)
```

/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:89: UserWarning:
The secret 'HF_TOKEN' does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (<https://huggingface.co/settings/tokens>), set it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
warnings.warn(
model.safetensors: 100%  188M/188M [00:52<00:00, 2.98MB/s]

▼ freeze and unfreeze layers

```
[ ] for param in model.parameters():  
    param.requires_grad = False
```

```
[ ] for name, param in model.named_parameters():  
    if "blocks.23" in name:  
        param.requires_grad = True
```

تعداد پارامترهای قابل آموزش:

▼ B - trainable parameters

```
[ ] trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)  
print(f'Number of trainable parameters: {trainable_params}')
```

Number of trainable parameters: 1775376

:Fine-Tune

▼ C - fine-tune

```
[ ] num_classes = len(classes)
    model.head = nn.Linear(model.head.in_features, num_classes)

[ ] criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=0.0001)
```

▼ train function

```
[ ] def train(model, trainloader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for inputs, labels in trainloader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * inputs.size(0)
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()
    epoch_loss = running_loss / len(trainloader.dataset)
    epoch_accuracy = 100. * correct / total
    return epoch_loss, epoch_accuracy
```

▼ validation function

```
[ ] def validate(model, testloader, criterion, device):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            running_loss += loss.item() * inputs.size(0)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()
    epoch_loss = running_loss / len(testloader.dataset)
    epoch_accuracy = 100. * correct / total
    return epoch_loss, epoch_accuracy
```

این مدل را نیز من در ۱۰ اپیاک بررسی کردم:

▼ training loop

```
[ ] model.to(device)

[ ] num_epochs = 10
    train_losses, train_accuracies = [], []
    val_losses, val_accuracies = [], []
    train_times, val_times = [], []
```

که نتیجه آموزش بدین صورت است:

```
for epoch in range(num_epochs):
    start_time = time.time()
    train_loss, train_accuracy = train(model, trainloader, criterion, optimizer, device)
    end_time = time.time()
    train_times.append(end_time - start_time)

    start_time = time.time()
    val_loss, val_accuracy = validate(model, testloader, criterion, device)
    end_time = time.time()
    val_times.append(end_time - start_time)

    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)
    val_losses.append(val_loss)
    val_accuracies.append(val_accuracy)

print(f"\n\n\n----- Epoch {epoch+1}/{num_epochs} -----")
print(f"Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.2f}%\n "
      f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_accuracy:.2f}%\n "
      f"Train Time: {train_times[-1]:.2f}s, Validation Time: {val_times[-1]:.2f}s\n")
```

و نتیجه هر اپیاک اینگونه شد برای این مدل:

-----Epoch 1/10-----

Train Loss: 0.2077, Train Accuracy: 93.27%

Validation Loss: 0.1199, Validation Accuracy: 95.92%

Train Time: 587.99s, Validation Time: 109.22s

-----Epoch 2/10-----

Train Loss: 0.0734, Train Accuracy: 97.65%

Validation Loss: 0.1157, Validation Accuracy: 96.11%

Train Time: 590.77s, Validation Time: 109.19s

-----Epoch 3/10-----

Train Loss: 0.0327, Train Accuracy: 99.06%

Validation Loss: 0.1044, Validation Accuracy: 96.80%

Train Time: 591.61s, Validation Time: 108.80s

-----Epoch 4/10-----

Train Loss: 0.0136, Train Accuracy: 99.65%

Validation Loss: 0.1221, Validation Accuracy: 96.39%

Train Time: 590.30s, Validation Time: 109.18

-----Epoch 5/10-----

Train Loss: 0.0060, Train Accuracy: 99.87%

Validation Loss: 0.1332, Validation Accuracy: 96.56%

Train Time: 590.78s, Validation Time: 109.11s

-----Epoch 6/10-----

Train Loss: 0.0069, Train Accuracy: 99.81%

Validation Loss: 0.1294, Validation Accuracy: 96.87%

Train Time: 591.22s, Validation Time: 109.21s

-----Epoch 7/10-----

Train Loss: 0.0045, Train Accuracy: 99.87%

Validation Loss: 0.1427, Validation Accuracy: 96.59%

Train Time: 591.13s, Validation Time: 108.97s

-----Epoch 8/10-----

Train Loss: 0.0041, Train Accuracy: 99.88%

Validation Loss: 0.1658, Validation Accuracy: 96.58%

Train Time: 590.05s, Validation Time: 109.14s

-----Epoch 9/10-----

Train Loss: 0.0048, Train Accuracy: 99.85%

Validation Loss: 0.1529, Validation Accuracy: 96.66%

Train Time: 591.03s, Validation Time: 108.97s

-----Epoch 10/10-----

Train Loss: 0.0037, Train Accuracy: 99.89%

Validation Loss: 0.1578, Validation Accuracy: 96.78%

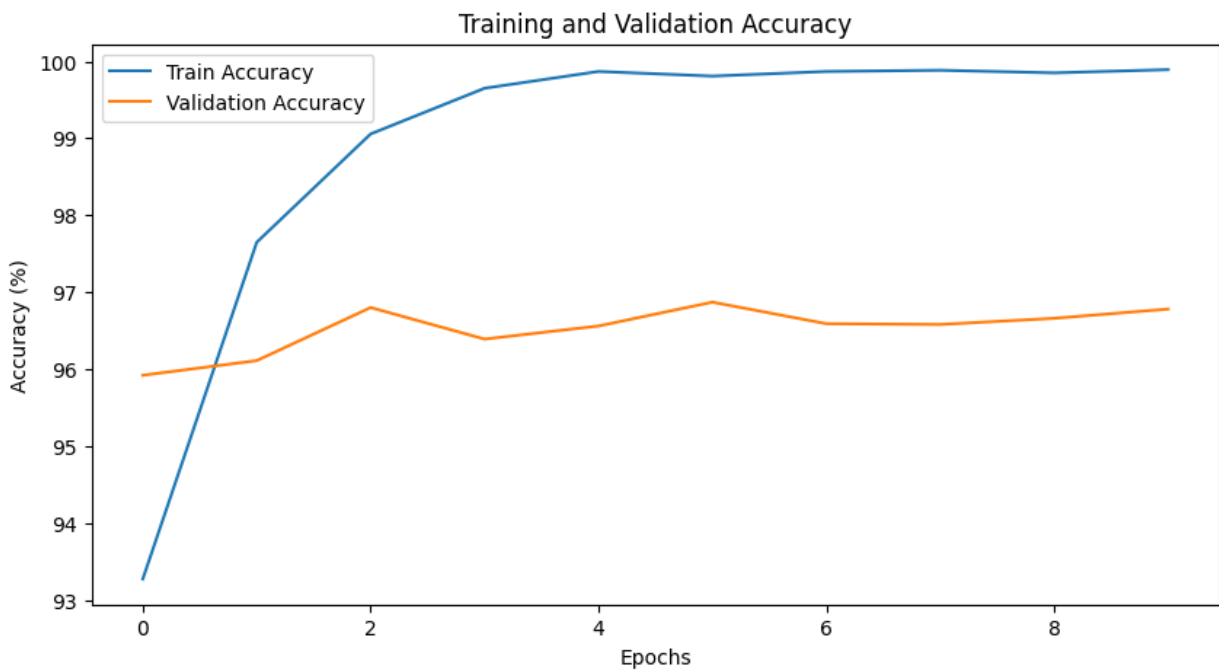
Train Time: 591.35s, Validation Time: 109.40s

و اینبار نمودارهای این مدل نیز بدین گونه در آمدند:

نمودار مقدار تابع هزینه به ازای هر اپاک:



مقدار دقت به ازای هر اپاک:



که در داده های آموزشی بهترین دقت ۹۹.۸۹٪ شده و در داده های اعتبارسنجی نیز دقت برابر ۹۶.۷۸٪ شد.

و میانگین زمان آموزش و اعتبارسنجی در هر اپاک بدین صورت شد:

E - calculate training and validation time (bonus)

```

[ ] total_train_time = sum(train_times)
    total_val_time = sum(val_times)
    average_train_time = total_train_time / num_epochs
    average_val_time = total_val_time / num_epochs

    print(f"Average training time per epoch: {average_train_time:.2f}s")
    print(f"Average validation time per epoch: {average_val_time:.2f}s")
  
```

Average training time per epoch: 590.62s
 Average validation time per epoch: 109.12s

۲.۵. مقایسه نتایج

Model	Model type	Trainable parameters	Validation accuracy (paper)	Validation accuracy (my results)	Train accuracy	Training time per epoch	Validation time per epoch
Densenet201	CNN	۲۳۵۲۱۰	94.757	۹۱.۴۴	۹۷.۷۲	۵۲۹۹.۵۹	۵۵۳.۱۲
DeiTBaseDistilled	transformer	۷۱۰۳۲۵۲	96.450	۹۶.۴۴	۹۹.۸۷	۶۲۶.۴۴	۱۰۸.۶۶
CaiTS24	transformer	۱۷۷۵۳۷۶	96.00	۹۶.۷۸	۹۹.۸۹	۵۹۰.۶۲	۱۰۹.۱۲

با توجه به اینکه تمامی مراحل را من سعی کردم تا عینا مشابه مقاله انجام بدهم و تنها نقطه تفاوت من با مقاله در تعداد ایپاک هایی بود که اجرا کردم، فلذا فکر می کنم دلیل اختلاف جزئی بدلیل عدم تساوی بودن تعداد ایپاکها باشد.