

به نام خدا



دانشگاه تهران



دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق
تمرین پنجم

علیرضا نجفی مطیعی	نام و نام خانوادگی	پرسش ۱
810100224	شماره دانشجویی	
پوریا ذره پرور قوچانی نژاد	نام و نام خانوادگی	پرسش ۲
۸۱۰۱۰۲۱۴۱	شماره دانشجویی	
1404.03.21	مهلت ارسال پاسخ	

فهرست

1	پرسش 1. طبقه‌بندی تصاویر با ViT
1	1. مقدمه
1	2. آماده‌سازی داده‌ها
3	3. آموزش مدل CNN
5	4. آموزش مدل ViT
8	5. تحلیل و نتیجه گیری
10	پرسش 2. Robust Zero-Shot Classification
10	1-۲. آشنایی با مدل CLIP و طبقه بندی تک ضرب و حملات خصمانه
11	1. جمع‌آوری داده‌ها:
12	2. فرآیند آموزش:
14	14. دفاع (Defenses)
14	14. زمینه یادگیری انتقالی (Transfer Learning)
19	2-۲. پیاده سازی و مقایسه روش‌های آموزش خصمانه

شکل‌ها

شکل 1. عنوان تصویر نمونه Error! Bookmark not defined.

جدول‌ها

جدول 1. عنوان جدول نمونه Error! Bookmark not defined.

پرسش ۱. طبقه‌بندی تصاویر با ViT

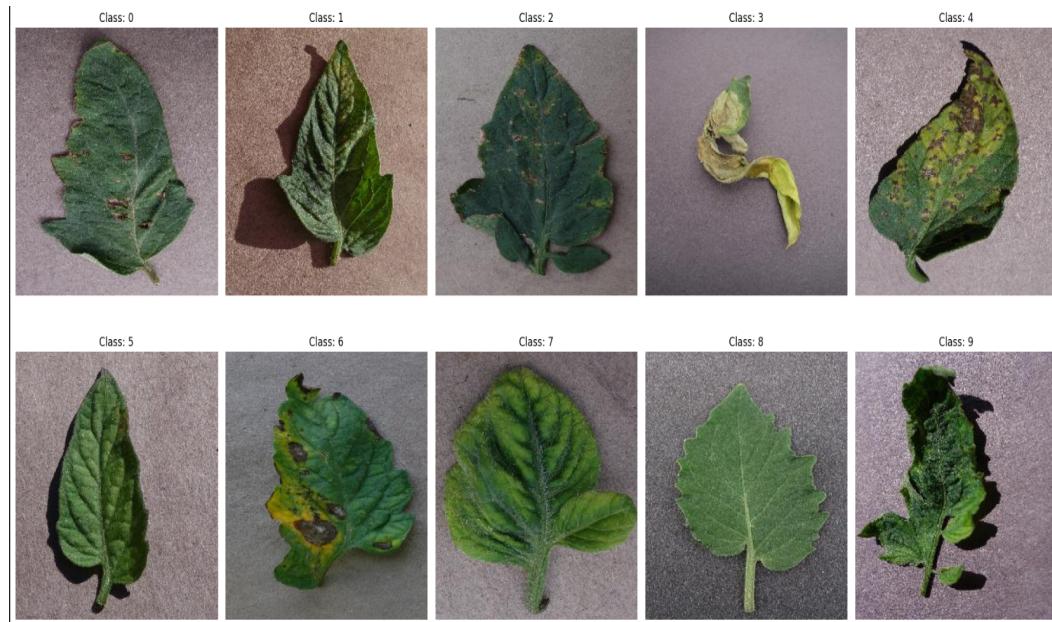
۱-۱. مقدمه

• مدل ViT با بهره‌گیری از مکانیسم خود توجهی رویکردی دگرگون کننده را معرفی می‌کند و به مدل اجازه می‌دهد تا بدون استخراج دستی بر ویژگی‌های مرتبط تمرکز کند. معماری آن نیز ذاتاً نیاز به استخراج خودکار و کارآمد ویژگی‌ها را برطرف می‌کند و به طور بالقوه نگرانی‌ها در مورد حذف اطلاعات حیاتی در فرآیند تشخیص را کاهش می‌دهد. همچنین ViT ذاتاً مکانیسم‌های توجه را از طریق مکانیسم‌های خود توجهی خود در بر می‌گیرد. برخلاف مدل‌های سنتی ViT بصورت پویا در طول پردازش به مناطق مختلف تصویر ورودی وزن اختصاص می‌دهد. این امر به چالش توزیع وزن برابر در ویژگی‌های عمیق می‌پردازد و به طور بالقوه تضمین می‌کند که ویژگی‌های مهم در طول طبقه‌بندی بیماری‌های برگ گوجه فرنگی توجه کافی را دریافت می‌کنند. در اصل مدل ViT، با مکانیسم‌های توجه و معماری منحصر به فرد خود، مسیری امیدوارکننده برای پیشبرد استخراج خودکار ویژگی‌ها و پرداختن به چالش‌های ناشی از رویکردهای سنتی فراهم می‌کند. این مدل با نیازهای در حال تحول یادگیری عمیق در زمینه تشخیص بیماری برگ گوجه فرنگی همسو است و راه حل‌های بالقوه‌ای را برای سوالات شناسایی شده در این زمینه ارائه می‌دهد.

• مدل‌های CNN با مجموعه داده‌های کوچک احتمالاً بهتر عمل می‌کنند زیرا بایاس‌های استقرایی داخلی آن‌ها (Inductive bias) به عنوان نوعی از regularization عمل می‌کنند. این به این معنی است که این مدل‌ها یکسری فرض‌های داخلی اولیه دارند که در پردازش تصاویر بسیار تاثیرگذارند. شبکه می‌داند که پیکسل‌های مجاور با هم اهمیت دارند و الگوهای می‌توانند در تصویر تغییر کنند. ViT‌ها معمولاً به داده بیشتری نیاز دارند تا آن بایاس‌های مشابه را از ابتدا پیدا کنند. بدون نمونه‌های آموزشی کافی ممکن است overfit شوند یا در ثبت الگوهای سطح پایین به طور قابل اعتمادی شکست بخورند. ViT یک تصویر را به دنبالهای از patch‌ها تبدیل می‌کند و از مکانیزم خود توجهی استفاده می‌کند تا هر پچ تصویر را با پچ دیگر آن مقایسه کند و بصورت ذاتی نمی‌داند که پچ ۱ در مجاورت پچ ۲ قرار گرفته یا ویژگی‌های محلی دارای اهمیت هستند. بنابراین این مدل باید تمام روابط فضایی و الگوهای ظاهری را کاملاً از خود دیتا بدست بیاورد. بنابراین زمانی که میزان دیتا کم باشد از نظر منطقی انتظار داریم تا CNN بهتر عمل کند اما لزوماً همیشه این اتفاق نخواهد افتاد.

۲-۱. آماده‌سازی داده‌ها

• داده‌ها در ده کلاس با لیبل‌های ۰ تا ۹ تقسیم شده‌اند که از هر کلاس یک داده را به نمایش می‌گذاریم:



شکل 1. نمونه‌ای از هر کلاس

- تعداد تصویر های مربوط به هر کلاس را بدست می آوریم که بصورت زیر است:

Class Label	Number of Images
0	1000
1	1000
2	1000
3	1000
4	1000
5	952
6	1000
7	1000
8	1000
9	373

شکل 2. جدول تعداد عکس‌های هر کلاس

مشاهده می شود که دو کلاس 5 و 9 تعداد عکس کمتری دارند که این مورد در کلاس 9 مشهودتر است بنابراین تعداد دادگان موجود متوازن نیستند. با توجه به اینکه داده‌ها بصورت عکس هستند و مسئله نیز مسئله classification است استفاده از data augmentation می‌تواند مفید باشد و آسیب جدی نیز به کیفیت تصاویر وارد نمی‌آورد (برای مثال تکنیک‌هایی مانند آینه کردن تصویر، چند درجه چرخاندن تصویر و کاهش میزان روشنایی آن). پس از اعمال این تکنیک توزیع به صورت زیر در می‌آید:

New, Balanced Class Distribution:	
label	
0	1000
1	1000
2	1000
3	1000
4	1000
5	1000
6	1000
7	1000
8	1000
9	1000

شکل 3. جدول دادگان بالанс شده

- استفاده بیشتر از data augmentation ممکن است مفید باشد و دقت را بالاتر ببرد اما فعلاً بدون آن پیش می‌رویم تا دقت مدل‌ها را بررسی کنیم. ابتدا بررسی می‌کنیم که ابعاد تصاویر موجود در دیتابست به چه صورت است و سپس طبق گفته مقاله که ورودی مدل ViT را 64×64 قرار داده، دادگان را resize می‌کنیم. همچنین ورودی اولیه مدل Inception-V3 بصورت 299×299 است که آن را هم بصورت 256×256 در می‌آوریم تا از دوباره resize کردن تصاویر خودداری کنیم:

```
Shape of a sample original image (Width, Height): (256, 256)
Resizing all images to 64x64... This may take a moment.
Resizing complete.
Shape of a sample image in 'resized_df' (Width, Height): (64, 64)
```

- در مقاله ده درصد به تست، و از 90 درصد باقیمانده 90 درصد آن برای آموزش و ده درصد برای اعتبارسنجی استفاده شده‌اند که در اینجا چون تست نداریم 80 درصد را به تست و 20 درصد را به اعتبارسنجی اختصاص می‌دهیم:

```
Data preparation complete! Here are the shapes for the resized (64x64) dataset:
x_train_resized shape: (8000, 64, 64, 3)
y_train_resized shape: (8000,)
x_val_resized shape: (2000, 64, 64, 3)
y_val_resized shape: (2000,)
```

3-1. آموزش مدل CNN

- بر خلاف مقاله که از وزن‌های ImageNet استفاده می‌کنیم در اینجا مدل را بصورت خام بارگذاری می‌کنیم. همچنین ورودی مدل را به 256×256 تغییر می‌دهیم تا مجبور نباشیم عکس‌ها را مجدداً resize کنیم و خروجی مدل را نیز مناسب برای ده کلاس تغییر می‌دهیم. خلاصه مدل ایجاد شده در زیر آمده است:

Custom InceptionV3 Model Summary:		
Model: "sequential_2"		
Layer (type)	Output Shape	Param #
inception_v3 (Functional)	(None, 6, 6, 2048)	21,802,784
global_average_pooling2d (GlobalAveragePooling2D)	(None, 2048)	0
dense (Dense)	(None, 10)	20,490

Total params: 21,823,274 (83.25 MB)
Trainable params: 21,788,842 (83.12 MB)
Non-trainable params: 34,432 (134.50 KB)

شکل 4. خلاصه مدل Inception-V3

این مدل یک مدل CNN است که برای این طراحی شده تا برخی مشکلات اساسی در CNN‌ها را بطرف کند. اینگونه کار می‌کند که بجای انتخاب کردن یک سایز برای فیلتر، کانولوشن‌ها و poolings‌های متفاوتی را بصورت موازی روی یک ورودی انجام می‌دهد. در نهایت نتیجه این شاخه‌های موازی در یک فیچر مپ بسیار عمیق concatenate می‌شود. در این ورژن از این مدل کانولوشن‌های بزرگتر با فاکتور گیری به کانولوشن‌های کوچک‌تر تبدیل می‌شوند. تابع خطای استفاده شده در مقاله categorical_crossentropy است که به زبان ساده یک تابع خطاست که تفاوت بین دو توزیع احتمال را اندازه‌گیری می‌کند یعنی احتمالات پیش‌بینی شده توسط مدل و لیبل‌های واقعی. برای مسئله‌های multi-class classification که هر داده تنها به یک کلاس تعلق دارد یک انتخاب مناسب این تابع خطاست. استفاده از این تابع خط نیازمند این است که لیبل‌های ما بصورت One-Hot کدگذاری شده باشند چرا که لایه نهایی softmax مدل نیز یک بردار احتمالی به اندازه تعداد کلاس‌ها به ما می‌دهد و این تابع خط یک مقدار کم را پیش‌بینی می‌کند اگر که دو توزیع شبیه هم باشند و یک مقدار بزرگ را اگر که تفاوت آن‌ها زیاد باشد. ساده شده فرمول آن بصورت $-\log(p)$ است که p احتمال این است که مدل به کلاس درست رسیده باشد.

به جای این تابع خط از sparse_categorical_crossentropy می‌توانیم استفاده کنیم که از نظر ریاضی با تابع خطای قلی یکسان است اما برای استفاده از آن نیازی نیست که لیبل‌ها وان‌هات شود و از لیبل اینتجر می‌توانیم استفاده کنیم.

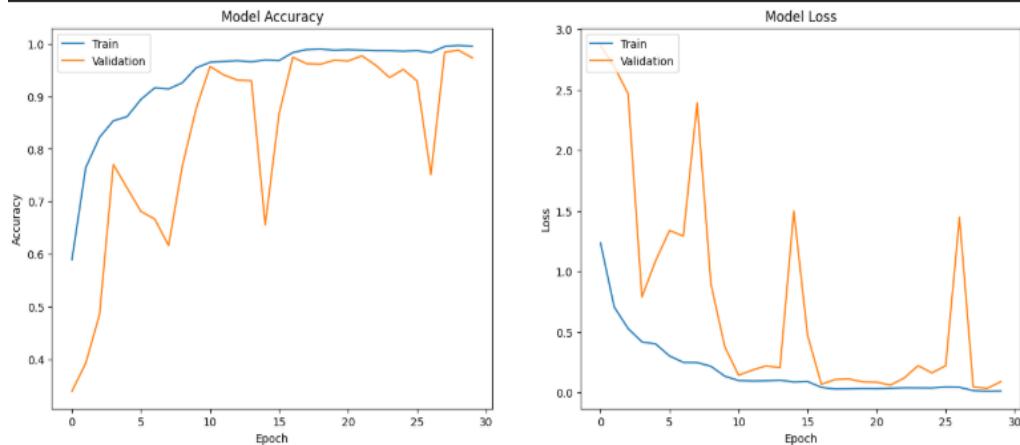
تابع خطای دیگری که می‌توان استفاده کرد Focal Loss است که ورژن تغییریافته cross-entropy است و به این صورت کار می‌کند که به صورت داینامیک وزن خطای مواردی را که به خوبی طبقه‌بندی شده‌اند را کاهش می‌دهد که باعث می‌شود مدل تلاش و تمرکز خود را روی مواردی که طبقه‌بندی آن‌ها سخت‌تر است بگذارد. البته از این تابع خط بیشتر در مواقعي استفاده می‌شود که دادگان نامتوازن باشند که ما در اینجا آن‌ها را متوازن کرده‌ایم. همچنین از توابع خطای Margin-Based Loss مانند Center Loss، Triplet Loss و ArcFace Loss می‌باشد. در این توابع خطای صرفاً توجه به پیش‌بینی درست هدف بهبود خود فضای ویژگی است.

• مدل را به مقدار 30 دوره آموزش می‌دهیم و به رسم نمودارهای گفته شده می‌پردازیم:

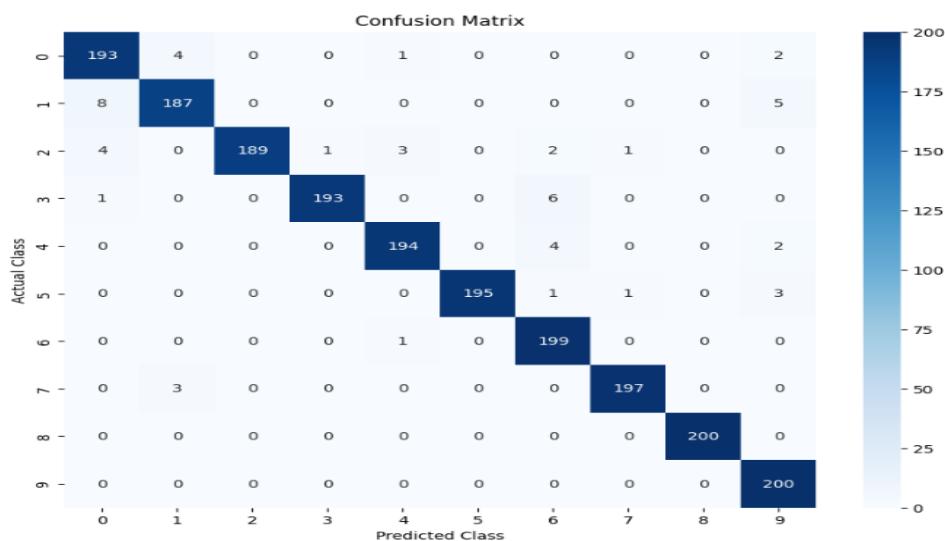
```

Epoch 28/30
250/250 79s 316ms/step - accuracy: 0.9932 - loss: 0.0234 - val_accuracy: 0.9845 - val_loss: 0.0460
- learning_rate: 1.2500e-04
Epoch 29/30
250/250 79s 318ms/step - accuracy: 0.9979 - loss: 0.0083 - val_accuracy: 0.9880 - val_loss: 0.0357
- learning_rate: 1.2500e-04
Epoch 30/30
250/250 79s 316ms/step - accuracy: 0.9933 - loss: 0.0164 - val_accuracy: 0.9735 - val_loss: 0.0907
- learning_rate: 1.2500e-04
Model training complete.

```



شکل ۵. نمودار دقت و خطا برای هر دو دسته داده در فرآیند آموزش



شکل 6. نمودار آشیانگی مدل برای داده validation

مشاهده می شود که مدل دقت بسیار بالایی دارد و با بالاتر بردن تعداد epoch ها احتمالاً دقت بالاتر نیز خواهد رفت. عملکرد این مدل و مدل بعدی را در بخش تحلیل و نتیجه گیری بیشتر بررسی خواهیم کرد.

4-1. آموزش مدل ViT

- با توجه به جدول 4 موجود در مقاله و همچنین توجه به اینکه به self و patch embedding attention نیاز داریم مدل را طراحی می کنیم و از پارامترهای گفته شده در مقاله استفاده می کنیم. بجای 4 سر attention از 8 یا 16 سر attention می کنیم تا هر سر با فضای با بعد بیشتری کار کند و بتواند رابطه های بهتری میان هر پچ تصویر پیدا کند. در نهایت خلاصه مدل طراحی

شده بصورت زیر است که کاملا با جدول آمده در مقاله هم از نظر ابعادی و هم از نظر تعداد پارامترها همخوانی دارد.

Layer (type)	Output Shape	Param #	Connected to
input_layer_5 (InputLayer)	(None, 64, 64, 3)	0	-
resizing_1 (Resizing)	(None, 80, 80, 3)	0	input_layer_5[0][0]
patches_1 (Patches)	(None, None, 192)	0	resizing_1[0][0]
patch_encoder_1 (PatchEncoder)	(None, 100, 64)	18,752	patches_1[0][0]
layer_normalization_5 (LayerNormalization)	(None, 100, 64)	128	patch_encoder_1[0][0]
multi_head_attention_2 (MultiHeadAttention)	(None, 100, 64)	132,672	layer_normalization_5[0][0], layer_normalization_5[0][0]
add_4 (Add)	(None, 100, 64)	0	multi_head_attention_2[0][0], patch_encoder_1[0][0]
layer_normalization_6 (LayerNormalization)	(None, 100, 64)	128	add_4[0][0]
dense_10 (Dense)	(None, 100, 128)	8,320	layer_normalization_6[0][0]
dropout_10 (Dropout)	(None, 100, 128)	0	dense_10[0][0]
dense_11 (Dense)	(None, 100, 64)	8,256	dropout_10[0][0]
dropout_11 (Dropout)	(None, 100, 64)	0	dense_11[0][0]
add_5 (Add)	(None, 100, 64)	0	dropout_11[0][0], add_4[0][0]
layer_normalization_7 (LayerNormalization)	(None, 100, 64)	128	add_5[0][0]
multi_head_attention_3 (MultiHeadAttention)	(None, 100, 64)	132,672	layer_normalization_7[0][0], layer_normalization_7[0][0]
add_6 (Add)	(None, 100, 64)	0	multi_head_attention_3[0][0], add_5[0][0]
layer_normalization_8 (LayerNormalization)	(None, 100, 64)	128	add_6[0][0]
dense_12 (Dense)	(None, 100, 128)	8,320	layer_normalization_8[0][0]
dropout_13 (Dropout)	(None, 100, 128)	0	dense_12[0][0]
dense_13 (Dense)	(None, 100, 64)	8,256	dropout_13[0][0]
dropout_14 (Dropout)	(None, 100, 64)	0	dense_13[0][0]
add_7 (Add)	(None, 100, 64)	0	dropout_14[0][0], add_6[0][0]
layer_normalization_9 (LayerNormalization)	(None, 100, 64)	128	add_7[0][0]
flatten_1 (Flatten)	(None, 6400)	0	layer_normalization_9[0][0]
dropout_15 (Dropout)	(None, 6400)	0	flatten_1[0][0]
dense_14 (Dense)	(None, 2048)	13,109,248	dropout_15[0][0]
dropout_16 (Dropout)	(None, 2048)	0	dense_14[0][0]
dense_15 (Dense)	(None, 1024)	2,098,176	dropout_16[0][0]
dropout_17 (Dropout)	(None, 1024)	0	dense_15[0][0]
dense_16 (Dense)	(None, 10)	10,250	dropout_17[0][0]

```
Total params: 15,535,562 (59.26 MB)
Trainable params: 15,535,562 (59.26 MB)
Non-trainable params: 0 (0.00 B)
```

شکل 7. خلاصه مدل ViT

هدف اولیه استفاده از این لایه این است که یک تصویر را حالت طبیعی و فرمت گرید فضایی خودش به دنبالهای از بردارها که یک Transformer می‌تواند بهمراه ترجمه کند. در حقیقت Transformer ها اول برای کاربردهای پردازش زبان طبیعی طراحی شده بودند که در آن دنباله ای از کلمات را پردازش می‌کنند. در حقیقت لایه Patch Embedding یک نوآوری مناسب است که اجزاء می‌دهد از آن‌ها در این کاربرد نیز استفاده کنیم. این پروسه از دو قدم اصلی تشکیل شده است:

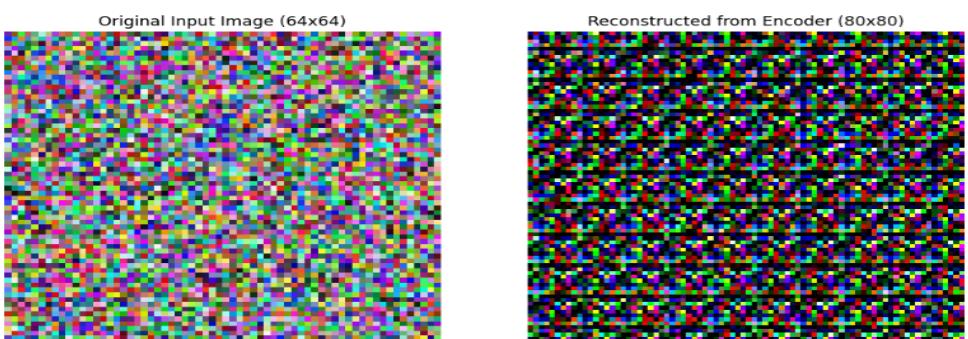
-1: تصویر ورودی در ابتدا به یک شبکه از پچ‌های بدون همپوشانی و مربعی تقسیم می‌شود. پس از آن هر پچ به یک بردار یک بعدی از مقادیر پیکسلی تبدیل می‌شود. در این مرحله تصویر به دنبالهای از بردارها تبدیل می‌شود.

-2: پس از آن این بردارها وارد یک شبکه Dense Embedding (Linear Projection) استاندارد می‌شوند. این مرحله داده پیکسلی خام را به یک فضای ویژگی پرمعنا تر و با بعد کمتر تبدیل می‌کند.

در نهایت یک positional embedding قابل یادگیری به هر patch embedding اضافه می‌شود. در حقیقت این کار اطلاعات فضایی مانند این که هر پچ در چه جایی از تصویر بوده را دوباره به عنوان داده به مدل تزریق می‌کند.

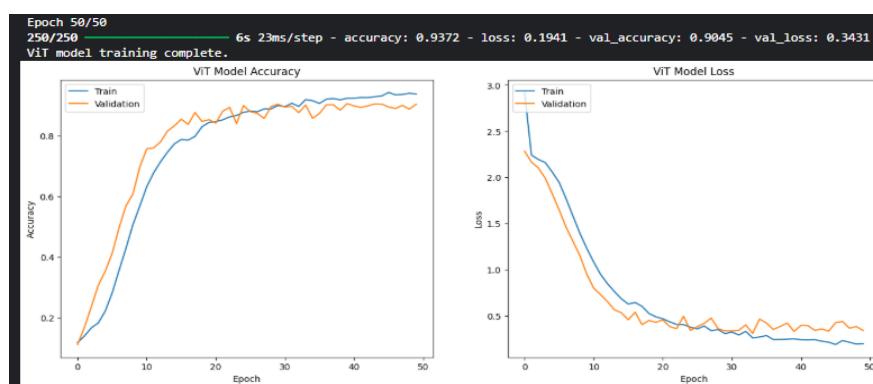
در اینجا از پچ‌های 8 در 8 برای تصاویر 64 در 64 استفاده می‌کنیم که در حالت کلی 64 پچ را ایجاد می‌کند. با کاهش اندازه هر پچ تعداد کلی پچ‌ها افزایش پیدا خواهد کرد که باعث می‌شود تا مدل بینشی با رزولوشن بالاتر از تصاویر دریافت کند که احتمالاً دقیق‌تر را بالاتر می‌برد اما باعث می‌شود تا هزینه محاسباتی ما بالاتر برود و مدل کنترل شود و به سخت‌افزار بهتری احتیاج پیدا کند. همچنین در این حالت خطر overfitting نیز با افزایش پارامترها افزایش پیدا خواهد کرد. افزایش اندازه هر پچ نیز دقیقاً بر عکس حالت بالا را نتیجه خواهد داد.

- یک قابلیت خروجی پیکسلی می‌گذاریم و یک نمونه از خروجی آن را روی یک تصویر الکی بررسی می‌کنیم:

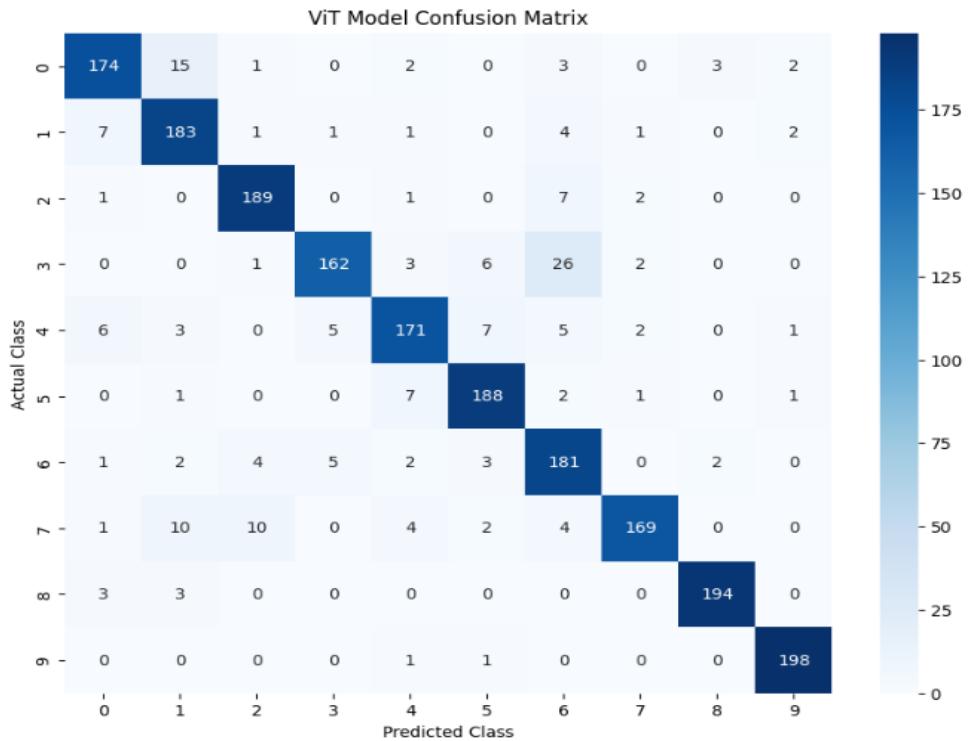


شکل 8. خروجی پیکسلی

- مدل را با 50 ایپاک آموزش می‌دهیم و نمودارهای خواسته شده را رسم می‌کنیم:



شکل 9. نمودار دقیق و خطای برای ViT



شکل 10. ماتریس آشتفتگی برای validation ViT

5-5. تحلیل و نتیجه گیری

- از classification_report برای هر دو مدل استفاده می‌کنیم تا مواردی مانند دقت و صحت را برای هر دو مدل داشته باشیم. همچنین تعداد پارامتر هر مدل در خلاصه آن آمده است که مشاهده شد مدل Inception-V3 دارای حدود 21 میلیون و 800 هزار پارامتر است درصورتیکه این مقدار برای مدل ViT در حدود 13 میلیون و 500 هزار پارامتر است. همچنین جزئیات دقت و صحت و همچنین معیارهایی مانند F1-Score دو مدل در زیر آمده است:

InceptionV3 Classification Report:				
	precision	recall	f1-score	support
Class 0	0.94	0.96	0.95	200
Class 1	0.96	0.94	0.95	200
Class 2	1.00	0.94	0.97	200
Class 3	0.99	0.96	0.98	200
Class 4	0.97	0.97	0.97	200
Class 5	1.00	0.97	0.99	200
Class 6	0.94	0.99	0.97	200
Class 7	0.99	0.98	0.99	200
Class 8	1.00	1.00	1.00	200
Class 9	0.94	1.00	0.97	200
accuracy			0.97	2000
macro avg	0.97	0.97	0.97	2000
weighted avg	0.97	0.97	0.97	2000
63/63		0s	6ms/step	

ViT Classification Report:				
	precision	recall	f1-score	support
Class 0	0.90	0.87	0.89	200
Class 1	0.84	0.92	0.88	200
Class 2	0.92	0.94	0.93	200
Class 3	0.94	0.81	0.87	200
Class 4	0.89	0.85	0.87	200
Class 5	0.91	0.94	0.92	200
Class 6	0.78	0.91	0.84	200
Class 7	0.95	0.84	0.90	200
Class 8	0.97	0.97	0.97	200
Class 9	0.97	0.99	0.98	200
accuracy			0.90	2000
macro avg	0.91	0.90	0.90	2000
weighted avg	0.91	0.90	0.90	2000

شکل 11. مقایسه دقت و صحت و سایر معیارها برای دو مدل

مشاهده شد که در اینجا مدل Inception از مدل ViT عملکرد بهتری داشت و در تمامی معیارها غیر از تعداد پارامترها از آن بهتر بود.

- برخلاف مقاله در اینجا ما از وزن‌های ImageNet استفاده نکردیم و مدل CNN را مجدداً و به صورت کامل روی دادگان خودمان آموزش دادیم که باعث شد با توجه به عمق زیاد این مدل ویژگی‌های منحصر به فرد Inception-V3 این مدل عملکرد بسیار خوبی از خود نشان دهد و اگر تعداد epoch ها را افزایش می‌دادیم احتمالاً Accuracy از این نیز بیشتر می‌شد. اما در این پیاده‌سازی مدل ViT دقتی که روی دادگان Validation داشت تقریباً با دقتی که مقاله از آن روی دادگان تست گزارش کرده بود اما از دقت روی داده Validation کمتر بود و در اینجا این مدل بهتر از مدل Inception عمل نکرد. با توجه به نکاتی که در بخش‌های قبل مطرح شد بدلیل عدم وجود بایاس‌های ذاتی اینگونه مدل‌ها به تعداد داده زیادی نیاز دارند. اگر تعداد داده ما بیشتر بود و یا از Data Augmentation برای همه کلاس‌ها استفاده می‌کردیم و داده را چند برابر می‌کردیم این مدل بهتر عمل می‌کرد. همچنین ذکر کردیم که کاهش اندازه پنج‌ها می‌تواند باعث افزایش دقت مدل شود. همچنین استفاده از تعداد epoch بیشتر، توابع خطأ با تمرکز بیشتر روی کلاس‌های دشوارتر، عمیق‌تر کردن مدل و همچنین افزایش لایه‌های Transformer که در اینجا برای اینکه مناسب پیاده‌سازی روی موبایل باشد دو بود از جمله مواردی است که می‌تواند باعث شود عملکرد این مدل بهبود یابد.

پرسش ۲. Robust Zero-Shot Classification

۱-۲. آشنایی با مدل CLIP و طبقه بندی تک ضرب و حملات خصمانه

FGSM یک روش تک مرحله‌ای برای تولید نمونه‌های تخصصی است. این روش از گرادیان‌های مدل نسبت به ورودی برای ایجاد اختلال‌های کوچک و نامحسوس استفاده می‌کند که باعث می‌شود مدل پیش‌بینی نادرستی انجام دهد.

هدف FGSM یافتن یک نمونه تخصصی x_a است که از تصویر اصلی X با افزودن یک اختلال کوچک دلتا ایجاد می‌شود به گونه‌ای که مدل پیش‌بینی نادرستی تولید کند در حالی که x_a از نظر بصری مشابه X باقی می‌ماند.

فرمول بهینه سازی :

X تصویر ورودی اصلی

y برچسب واقعی تصویر

L تابع زیان متنی - تصویری

$\nabla_x L$ گرادیان زیان نسبت به تصویر ورودی

$\text{Sign}()$ تابع علامت که جهت گرادیان را استخراج می‌کند.

ϵ اندازه گام یا شعاع اختلال که میزان تغییر را محدود می‌کند.

فرآیند بهینه سازی :

۱ - محاسبه گرادیان زیان L نسبت به تصویر ورودی X

۲ - استخراج جهت گرادیان با استفاده از تابع sign

۳ - افروden اختلال ϵ . $\text{sign}(\nabla_x L)$ به تصویر اصلی برای تولید x_a

ویژگی ها :

سرعت : FGSM به دلیل تک مرحله‌ای بودن بسیار سریع است و نیازی به بهینه سازی تکراری ندارد.

محدودیت ها :

به دلیل سادگی حملات FGSM ممکن است به اندازه روش‌های تکراری مانند PGD قوی نباشند. زیرا تنها یک گام در جهت گرادیان بر می‌دارند.

PGD به صورت تکراری گرادیان را محاسبه کرده و اختلال را به روزرسانی می‌کند، در حالی که اطمینان حاصل می‌کند که اختلال در یک محدوده مشخص باقی می‌ماند. این روش به دنبال یافتن قوی‌ترین نمونه تخصصی در یک محدوده محدود است.

قوی‌تر از FGSM، زیرا چندین مرحله بهینه‌سازی انجام می‌دهد. می‌تواند نمونه‌های تخصصی پیچیده‌تر و مؤثرتری تولید کند. انعطاف‌پذیر است و با تنظیم تعداد مراحل و اندازه گام می‌توان شدت حمله را کنترل کرد.

محاسباتی سنگین‌تر از FGSM است، زیرا نیاز به چندین محاسبه گرادیان و بهروزرسانی دارد. زمان بر است، بهویژه برای تعداد مراحل بالا.

رمزگذار تصویر:

معمولًا از یک معماری مبتنی بر ResNet یا Vision Transformer (ViT) استفاده می‌شود. برای ViT، تصویر به بیچهای کوچک تقسیم شده و به توکن‌های ورودی تبدیل می‌شود. این توکن‌ها از طریق لایه‌های ترسنفورمر پردازش شده و یک بردار ویژگی کلی تولید می‌کنند.

در مقاله، از معماری ViT-B/32 استفاده شده است، که شامل ۳۲*۳۲ پیچ برای ورودی است. رمزگذار متن :

معمولًا یک Transformer مبتنی بر متن مانند BERT یا GPT است که متن ورودی توضیحات یا برچسب‌ها را به یک بردار ویژگی تبدیل می‌کند.

هر دو رمزگذار تصویر و متن، ویژگی‌ها را به یک فضای برداری مشترک با ابعاد یکسان (معمولًا ۵۱۲ یا ۷۶۸ بعدی) نگاشت می‌کنند.

شباهت بین ویژگی‌های تصویر و متن با استفاده از شباهت کسینوسی (Cosine Similarity) محاسبه می‌شود.

در زمان تست، CLIP از شباهت کسینوسی بین ویژگی تصویر و ویژگی‌های متنی برای چندین کلاس استفاده می‌کند تا پیش‌بینی کند کدام متن (کلاس) با تصویر مطابقت دارد.

برای مثال، برای یک تصویر، شباهت کسینوسی با متن‌هایی مانند "A photo of a cat" و "A photo of a dog" غیره محاسبه شده و کلاسی با بالاترین شباهت انتخاب می‌شود.

تابع زیان CLIP به گونه‌ای طراحی شده است که ویژگی‌های تصویر و متن مربوط به هم (جفت‌های مثبت) را به هم نزدیک کند و جفت‌های غیرمرتبط (جفت‌های منفی) را از هم دور کند. این فرآیند به صورت متقابل برای هر دو جهت تصویر-به-متن و متن-به-تصویر انجام می‌شود.

آموزش کنتراستیو یک روش یادگیری خودناظارتی (self-supervised) است که برای یادگیری نمایش‌های مشترک بین دو نوع داده (در اینجا تصویر و متن) استفاده می‌شود. در CLIP، این روش برای پیش‌آموزش مدل روی مجموعه داده‌های عظیم تصویر-متن (مانند مجموعه داده‌های وب) به کار می‌رود.

مراحل آموزش در CLIP

۱. جمع‌آوری داده‌ها :

CLIP روی مجموعه داده‌های عظیم جفت‌های تصویر-متن آموزش می‌بیند .

- هر جفت شامل یک تصویر و یک توضیح متنی مرتبط (مانند کپشن یا برچسب) است.

۲. فرآیند آموزش :

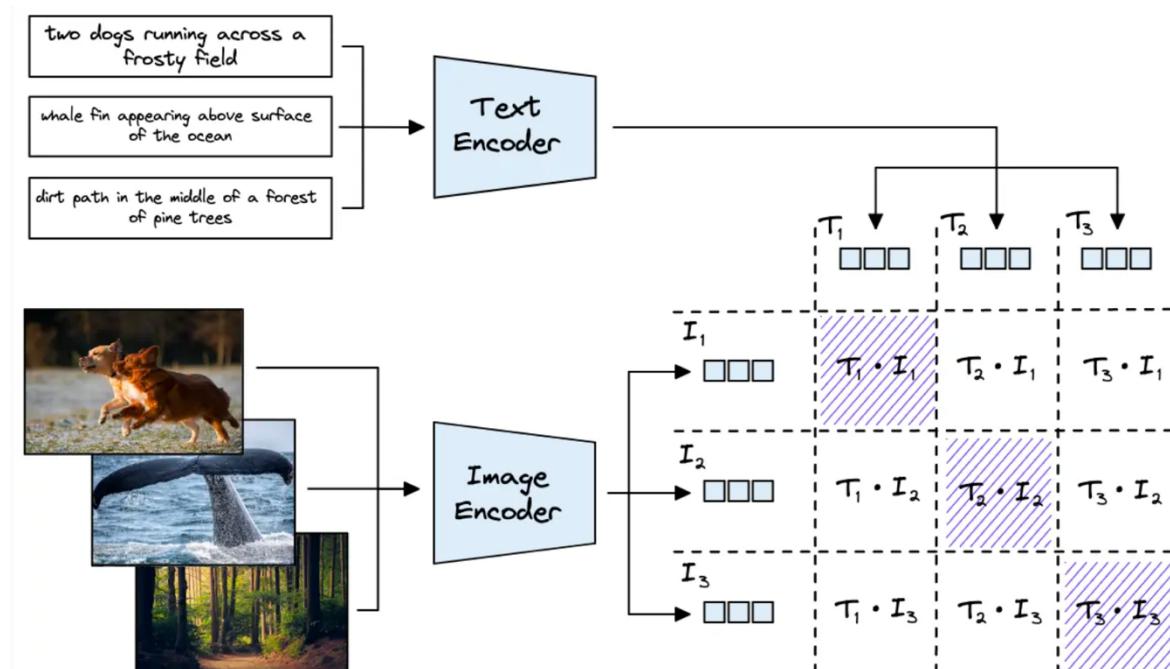
- برای هر دسته (batch) از جفت تصویر-متن :
- تصاویر به رمزگذار تصویر و متن‌ها به رمزگذار متن وارد می‌شوند.
- ویژگی‌های تصویر و متن استخراج شده و شباهت‌های کسینوسی محاسبه می‌شوند.
- تابع زیان کنتراستیو برای به حداقل رساندن شباهت جفت‌های مثبت و به حداقل رساندن شباهت جفت‌های منفی بهینه می‌شود.
- بهینه‌سازی با استفاده از الگوریتم‌هایی مانند SGD یا Adam انجام می‌شود.

ویژگی‌های آموزش :

- خودنظراتی: نیازی به برچسب‌های دقیق مانند one-hot labels نیست؛ تنها جفت‌های تصویر-متن مورد نیاز است.
- تعمیم‌پذیری صفر-شات: آموزش کنتراستیو باعث می‌شود مدل بتواند بدون آموزش مجدد روی وظایف جدید (مانند طبقه‌بندی صفر-شات) عمل کند.
- تراز چندوجهی Multimodal Alignment : ویژگی‌های تصویر و متن در یک فضای مشترک تراز می‌شوند، که امکان مقایسه مستقیم آن‌ها را فراهم می‌کند.

.۳

طبقه‌بندی عادی شامل آموزش مدل با چندین نمونه از هر کلاس است، که معمولاً به صدها یا هزاران تصویر نیاز دارد. این روش برای سناریوهایی مناسب است که داده‌های کافی موجود است.



شکل ۱۲. طبقه‌بندی تک‌ضرب CLIP

طبقه‌بندی تک ضرب بخشی از دسته‌بندی یادگیری با داده‌های کم (Few-Shot Learning) است، جایی که مدل باید بتواند کلاس‌های جدیدی را که در طول آموزش دیده نشده‌اند، با استفاده از اطلاعات مکمل مانند توصیفات متنی یا ویژگی‌های سماتیک شناسایی کند.

این روش معمولاً بر اساس مقایسه ویژگی‌های تصویر جدید با توصیفات متنی کلاس‌ها انجام می‌شود. به عنوان مثال، برای شناسایی یک شیء ناشناخته، ممکن است فقط متنی آن (مانند "عکسی از یک فیل") موجود باشد. تحقیقات نشان می‌دهد که این روش برای موقعی مفید است که داده‌های آموزشی برای کلاس‌های جدید موجود نباشد، مانند شناسایی اشیاء ناشناخته در تصاویر.

- غیرمرتبط را کمینه کند، با استفاده ازتابع ضرر کنترasti.(Contrastive Loss)

طبقه‌بندی تک ضرب با CLIP :

- برای طبقه‌بندی یک تصویر، کاربر باید لیستی از کلاس‌ها به صورت توصیفات متنی فراهم کند (مثلاً "عکسی از گربه"، "عکسی از سگ")
- CLIP تصویر ورودی را به بردار تصویر تبدیل می‌کند، و هر کدام از توصیفات متنی را به بردارهای متنی تبدیل می‌کند.
- سپس، شباهت مثلاً شباهت زاویه‌ای یا Cosine Similarity بین بردار تصویر و هر کدام از بردارهای متنی محاسبه می‌شود.
- کلاسی که بیشترین شباهت را دارد، به عنوان پیش‌بینی انتخاب می‌شود. این فرآیند به مدل اجازه می‌دهد تصاویر را به کلاس‌هایی که هرگز در طول آموزش دیده نیست، طبقه‌بندی کند.

تفاوت‌های کلیدی بین طبقه‌بندی عادی و تک ضرب (zero-shot) در تعداد داده‌های آموزشی و روش یادگیری است. برای طبقه‌بندی تک ضرب طراحی شده و از ترکیب کدگرهای تصویر و متن برای مقایسه شباهت استفاده می‌کند.

.۴

حملات جعبه سفید :

در حملات جعبه سفید، حمله کننده دسترسی کامل به مدل دارد، از جمله معماری مدل، پارامترها (وزن‌ها)، و داده‌های آموزشی. این سطح از دسترسی به حمله کننده اجازه می‌دهد نمونه‌های خصمانه‌ای ایجاد کند که به طور خاص برای سوءاستفاده از ضعف‌های مدل طراحی شده باشند.

با داشتن دسترسی به گرادیان‌های مدل نسبت به ورودی، حمله کننده می‌تواند از تکنیک‌های بهینه‌سازی استفاده کند تا اختلالاتی پیدا کند که خطای مدل را بیشینه کند.

روش‌های رایج شامل Fast Gradient Sign Method (FGSM) و نسخه‌های تکراری آن مانند I-FGSM هستند. این روش‌ها مستقیماً از گرادیان‌های مدل برای تولید نمونه‌های خصمانه استفاده می‌کنند.

حملات جعبه سیاه :

در حملات جعبه سیاه، حمله کننده دسترسی محدود یا هیچ دسترسی‌ای به جزئیات داخلی مدل ندارد. او ممکن است فقط به ورودی‌ها و خروجی‌های مدل دسترسی داشته باشد و شاید برخی اطلاعات در مورد رفتار مدل، اما نه پارامترها یا معماری آن.

با این حال، حملات جعبه سیاه می‌توانند با استفاده از قابلیت انتقال (Transferability) نمونه‌های خصمانه موفق باشند. قابلیت انتقال به این معناست که نمونه‌های خصمانه‌ای که برای یک مدل خاص ایجاد شده‌اند، گاهی اوقات می‌توانند مدل‌های دیگر را نیز فریب دهند، حتی اگر مدل‌ها متفاوت باشند.

برای انجام حملات جعبه سیاه، حمله کننده ممکن است یک مدل جایگزین (Substitute Model) روی داده‌های مشابه آموزش دهد و از آن برای تولید نمونه‌های خصمانه استفاده کند، با این امید که این نمونه‌ها به مدل هدف منتقل شوند.

روش دیگر، استفاده از روش‌های مبتنی بر پرس‌وجو است، که در آن حمله‌کننده مدل هدف را با ورودی‌های مختلف پرس‌وجو می‌کند و از خروجی‌ها برای هدایت تولید نمونه‌های خصمانه استفاده می‌کند.

علاوه بر این، تحقیقات نشان می‌دهد که در تنظیمات جعبه سیاه، روش‌های یک مرحله‌ای مانند FGSM و Targeted FGSM (T-FGSM) می‌توانند مؤثرتر از روش‌های تکراری باشند، زیرا روش‌های تکراری ممکن است به مدل جایگزین بیش‌برازش کنند، که قابلیت انتقال را کاهش می‌دهد.

اطلاعات مورد نیاز :

- در حملات جعبه سفید، حمله‌کننده نیاز به دانستن کامل معماری مدل، پارامترها، و داده‌های آموزشی دارد.
- در حملات جعبه سیاه، فقط به ورودی‌ها و خروجی‌های مدل نیاز است و اطلاعات داخلی مدل لازم نیست.

: (Effectiveness) موفقیت

- حملات جعبه سفید معمولاً موفق‌تر هستند زیرا حمله‌کننده می‌تواند نمونه‌های خصمانه را دقیقاً با ضعف‌های مدل هماهنگ کند.
- حملات جعبه سیاه کمتر موفق هستند، اما همچنان تهدید جدی هستند، به ویژه اگر نمونه‌های خصمانه قابلیت انتقال بالایی داشته باشند.

: (Practicality) عملیاتی بودن

- حملات جعبه سفید کمتر عملی هستند زیرا دسترسی به جزئیات داخلی مدل در سناریوهای واقعی، به ویژه برای مدل‌های مالکیتی یا ابری، دشوار است.
- حملات جعبه سیاه عملی‌تر هستند زیرا فقط نیاز به دسترسی به API یا رابط مدل دارند، که اغلب عمومی است.

: (Defenses) دفاع

- برای حملات جعبه سفید، دفاع‌ها ممکن است شامل آموزش خصمانه (Adversarial Training) باشد، که مدل را با نمونه‌های تمیز و خصمانه آموزش می‌دهد، یا روش‌هایی مانند ماسک‌گذاری گرادیان (Gradient Masking)، هرچند این روش‌ها ممکن است دور زده شوند.
- برای حملات جعبه سیاه، دفاع‌ها ممکن است روی شناسایی نمونه‌های خصمانه از طریق آزمون‌های آماری یا استفاده از روش‌های انسامبل تمرکز کنند.

: (Transfer Learning) زمینه یادگیری انتقالی

تحقیقات نشان می‌دهد که تنظیم مجدد (Fine-Tuning) در یادگیری انتقالی می‌تواند مقاومت مدل در برابر حملات جعبه سفید FGSM را افزایش دهد. تنظیم مجدد مقاومت مدل را در برابر حملات جعبه سفید افزایش می‌دهد.

با این حال، تنظیم مجدد می‌تواند قابلیت انتقال نمونه‌های خصمانه را در حملات جعبه سیاه نیز افزایش دهد، که باعث آسیب‌پذیری بیشتر مدل‌ها در این حملات می‌شود. این مقاله همچنین یک معیار جدید برای ارزیابی قابلیت انتقال نمونه‌های خصمانه بین مدل منبع و هدف پیشنهاد می‌دهد و نشان می‌دهد که نمونه‌های خصمانه با استفاده از تنظیم مجدد قابل انتقال‌تر هستند.

منابع محاسباتی (Computational Resources) :

- حملات جعبه سفید معمولاً نیاز به منابع محاسباتی کمتری دارند زیرا حمله کننده می‌تواند گرادیان‌ها را مستقیماً با استفاده از پارامترهای مدل محاسبه کند.
- حملات جعبه سیاه ممکن است نیاز به منابع بیشتری داشته باشند، به ویژه اگر حمله کننده بخواهد یک مدل جایگزین آموزش دهد یا تعداد زیادی پرس‌وجو به مدل هدف انجام دهد.

قابلیت شناسایی (Detectability) :

- نمونه‌های خصمانه در حملات جعبه سفید می‌توانند بسیار نازک و دشوارالشناسایی باشند زیرا به طور دقیق با مدل هماهنگ می‌شوند.
- در حملات جعبه سیاه، نمونه‌های خصمانه ممکن است کمتر نازک باشند و به دلیل عدم هماهنگی کامل با مدل، قابل شناسایی‌تر باشند، به ویژه با استفاده از روش‌های تشخیص ناهنجاری.

.۵

حملات انتقالی (Transferability Attacks) :

حملات انتقالی زمانی رخ می‌دهند که نمونه‌های خصمانه تولیدشده برای یک مدل خاص (معمولًاً یک مدل جایگزین) روی مدل‌های دیگر نیز مؤثر باشند، حتی اگر معماری یا داده‌های آموزشی آن‌ها متفاوت باشد. این ویژگی به ویژه در سناریوهای جعبه سیاه اهمیت دارد، زیرا مهاجم نیازی به دسترسی مستقیم به مدل هدف ندارد.

.۵

دلایل تهدید جدی‌تر بودن حملات انتقالی در دنیای واقعی نسبت به حملات جعبه سفید:

عدم نیاز به دسترسی به مدل هدف :

در دنیای واقعی، دسترسی به معماری و وزن‌های مدل (مانند حملات جعبه سفید) معمولاً غیرممکن است، زیرا شرکت‌ها این اطلاعات را محروم‌انه نگه می‌دارند. حملات انتقالی نیازی به این دسترسی ندارند و با استفاده از یک مدل جایگزین که به راحتی ساخته است، می‌توانند به مدل هدف حمله کنند.

واقع‌گرایی در سناریوهای عملی :

در بسیاری از کاربردهای واقعی (مانند سیستم‌های تشخیص چهره، خودروهای خودران، یا فیلترهای محتوای آنلاین)، مهاجمان تنها به خروجی‌های مدل دسترسی دارند (مانند API یا پاسخ‌های سیستمی). حملات انتقالی این امکان را فراهم می‌کنند که نمونه‌های خصمانه در یک محیط کنترل شده (مانند مدل جایگزین) تولید شوند و سپس روی مدل هدف آزمایش شوند.

مشکلات در دفاع :

دفاع در برابر حملات انتقالی دشوارتر است، زیرا این حملات از ویژگی‌های مشترک بین مدل‌های مختلف (مانند شباهت‌های آموخته شده در ویژگی‌های داده) بهره می‌برند. تکنیک‌های دفاعی مانند آموزش خصمانه (adversarial training) ممکن است در برابر حملات جعبه سفید مؤثر باشند، اما در برابر حملات انتقالی که از مدل‌های دیگر منتقل می‌شوند، اغلب ناکارآمد هستند.

گسترش‌پذیری تهدید :

حملات انتقالی می‌توانند به صورت گستردۀ روی مدل‌های مختلف اعمال شوند. برای مثال، یک نمونه خصمانه تولیدشده برای یک مدل تشخیص تصویر می‌تواند روی مدل‌های دیگر با معماری متفاوت نیز کار کند، که این امر تهدید را برای سیستم‌های گستردۀتر (مانند سیستم‌های امنیتی یا پزشکی) افزایش می‌دهد.

سادگی نسبی برای مهاجم :

مهاجم می‌تواند با استفاده از مدل‌های متن‌باز یا مدل‌هایی با معماری مشابه، نمونه‌های خصمانه تولید کند و آن‌ها را بدون نیاز به دانش عمیق از مدل هدف به کار ببرد. این امر حملات را برای مهاجمان با منابع محدود نیز قابل اجرا می‌کند.

تأثیر در سیستم‌های حیاتی :

در کاربردهای حساس مانند خودروهای خودران یا تشخیص پزشکی، یک نمونه خصمانه انتقالی می‌تواند عواقب جدی مانند تصادف یا تشخیص نادرست بیماری ایجاد کند. این تهدید بهویژه در جعبه سیاه خطرناک است، زیرا مهاجم می‌تواند بدون دسترسی مستقیم به مدل، آسیب وارد کند.

.۶

مراحل تنظیم دقیق : LoRA

انتخاب مدل پایه :

ابتدا یک مدل از پیش آموزش دیده انتخاب می‌شود. این مدل معمولاً شامل میلیون‌ها یا میلیاردها پارامتر است که تنظیم دقیق همه آن‌ها هزینه‌بر است.

اضافه کردن ماتریس‌های کم‌مرتبه (Low-Rank Matrices) :

در LoRA، به جای تغییر تمام وزن‌های مدل (W)، دو ماتریس کوچک‌تر A و B با رتبه کم به لایه‌های خاص (مانند لایه‌های توجه یا خطی) اضافه می‌شوند. این ماتریس‌ها تغییرات کوچک در وزن‌های اصلی را مدل می‌کنند.

$$\Delta W = A \times B$$

انتخاب لایه‌های هدف:

معمولتاً لایه‌های خاصی از مدل (مانند لایه‌های توجه در Transformer یا لایه‌های خطی) برای اعمال LoRA انتخاب می‌شوند. این انتخاب به نوع مدل و وظیفه موردنظر بستگی دارد.

آموزش ماتریس‌های LoRA

داده‌های تنظیم دقیق (مانند مجموعه داده‌ای خاص برای یک وظیفه یا سبک خاص) جمع‌آوری می‌شود.

ماتریس‌های A و B با استفاده از الگوریتم‌های بهینه‌سازی مانند Adam و با بهروزرسانی گرادیان‌ها آموزش داده می‌شوند.

ادغام و استفاده از مدل:

پس از آموزش، ماتریس‌های A و B می‌توانند با وزن‌های اصلی ادغام شوند.

یا به صورت جداگانه ذخیره شوند تا مدل سبک‌تر بماند.

$$B \times A + W = W'$$

سه علت استفاده از LoRA نسبت به روش‌های دیگر:

کاهش هزینه محاسباتی و مصرف حافظه :

در روش‌های سنتی تنظیم دقیق، تمام پارامترهای مدل (که ممکن است میلیاردها پارامتر باشد) به روزرسانی می‌شوند، که نیازمند منابع محاسباتی و حافظه زیادی است LoRA با تمرکز بر به روزرسانی ماتریس‌های کم‌مرتبه A و B به جای کل وزن‌ها، تعداد پارامترهای قابل تنظیم را به شدت کاهش می‌دهد. برای مثال، در یک مدل با ۱ میلیارد پارامتر، LoRA ممکن است تنها چند میلیون پارامتر را تنظیم کند.

جلوگیری از بیش‌برازش :

تنظیم دقیق کل مدل ممکن است به بیش‌برازش منجر شود، به‌ویژه زمانی که داده‌های تنظیم دقیق محدود هستند. LoRA با محدود کردن تغییرات به ماتریس‌های کم‌مرتبه، از تغییرات بیش از حد در مدل جلوگیری می‌کند و تعیین‌پذیری مدل را حفظ می‌کند.

انعطاف‌پذیری و مقیاس‌پذیری:

امکان ذخیره ماتریس‌های A و B به صورت جداگانه را فراهم می‌کند، که باعث می‌شود چندین تنظیم دقیق برای وظایف مختلف (مانند سبک‌های مختلف هنری یا زبان‌های تخصصی) به راحتی قابل اعمال و تعویض باشند. این برخلاف روش‌های سنتی است که نیاز به ذخیره کل مدل تنظیم‌شده دارد.

این ویژگی LoRA را برای کاربردهای چندوظیفه‌ای (مانند تولید تصاویر با سبک‌های مختلف یا پاسخ به سوالات در حوزه‌های مختلف) مناسب می‌کند و فضای ذخیره‌سازی را کاهش می‌دهد.

. ۷

مقاله اول Sigmoid Loss for Language Image Pre-Training

SigLIP:تابع زیان سیگموئید برای پیش‌آموزش زبان-تصویر

پژوهش ارائه شده تحت عنوان SigLIP، یک تابع زیان سیگموئید (pairwise Sigmoid loss) زوجی (pairwise) ساده را برای پیش‌آموزش زبان-تصویر پیشنهاد می‌کند. این رویکرد در نقطه مقابل یادگیری مقابله‌ای استاندارد قرار می‌گیرد که مبتنی بر نرمال‌سازی سافت‌مکس (softmax normalization) است و در مدل اصلی CLIP به کار گرفته شده بود. ویژگی برجسته تابع زیان سیگموئید این است که منحصراً بر روی زوچهای تصویر-متن عمل می‌کند و برای فرآیند نرمال‌سازی، نیازی به یک دید سراسری (global view) از شباهت‌های زوجی در کل بچ (batch) ندارد. این خصوصیت منجر به مزایای قابل توجهی در زمینه مقیاس‌پذیری و کارایی محاسباتی می‌شود. به طور مشخص، این تابع زیان امکان افزایش بیشتر اندازه بچ را فراهم می‌آورد و همزمان، در اندازه‌های بچ کوچکتر نیز عملکرد بهتری نسبت به تابع زیان مبتنی بر سافت‌مکس از خود نشان می‌دهد.

تابع زیان SigLIP به جای استفاده از سافت‌مکس، از تابع سیگموئید برای هر زوج تصویر-متن به صورت مستقل استفاده می‌کند. هدف این است که برای زوچهای مثبت، خروجی تابع سیگموئید به مقدار ۱ نزدیک شود و برای زوچهای منفی (تصویر و متن نامرتب)، این خروجی به ۰ میل کند. نکته حائز اهمیت این است که این رویکرد "نیازی به هیچ عملیاتی در کل بچ ندارد". این ویژگی، پیاده‌سازی توزیع شده تابع زیان را به طور قابل ملاحظه‌ای ساده‌تر کرده و کارایی محاسباتی را افزایش می‌دهد. علاوه بر این، تابع زیان سیگموئید به طور مفهومی "اندازه بچ را از تعریف وظیفه جدا می‌کند". این در حالی است که در تابع زیان مبتنی بر سافت‌مکس، تعداد نمونه‌های منفی موثر که در فرآیند یادگیری نقش دارند، مستقیماً به اندازه بچ وابسته است. از منظر مصرف منابع نیز، SigLIP نظر حافظه کارآمدتر است و معمولاً به حافظه کمتری نسبت به تابع زیان سافت‌مکس نیاز دارد.

یکی از یافته‌های کلیدی در مورد SigLIP این است که "در اندازه‌های بچ کوچکتر (کمتر از ۱۶ ھزار) به طور قابل توجهی بهتر از تابع زیان سافت‌مکس عمل می‌کند." این ویژگی، مقاومت مدل را در شرایطی که منابع محاسباتی محدود هستند و امکان استفاده از بچهای بسیار بزرگ فراهم نیست، افزایش می‌دهد. مدل‌هایی که می‌توانند با بچهای کوچکتر به طور موثر آموزش بینند.

آموزش بر روی مجموعه داده‌های بزرگتر و متنوع‌تر، یکی از مسیرهای کلیدی برای افزایش قابلیت تعمیم‌پذیری و در نتیجه مقاومت مدل در برابر داده‌های نادیده و خارج از توزیع است. سادگی پیاده‌سازی توزیع شده نیز، احتمال بروز خطاهای بالقوه در سیستم‌های محاسباتی پیچیده را کاهش می‌دهد. به عبارت دیگر، از آنجایی که SigLIP کارآمدتر از سافت‌مکس است، اجازه می‌دهد تا در زمان و با منابع محاسباتی یکسان، داده‌های بیشتری پردازش شوند یا مدل برای دوره‌های (epochs) بیشتری آموزش بینند.

تابع زیان سیگموئید، با تمرکز بر روی هر زوج تصویر-متن به صورت مجزا، ممکن است نسبت به تابع زیان سافت‌مکس که به شدت به کیفیت و توزیع نمونه‌های منفی درون بچ وابسته است، مقاومت بیشتری در برابر "نمونه‌های منفی سخت" یا "نمونه‌های منفی نویزی" نشان دهد. در تابع زیان سافت‌مکس، یک نمونه منفی که بسیار شبیه به نمونه مثبت است (اما به اشتباه به عنوان منفی در نظر گرفته شده یا ذاتاً گمراه کننده است) می‌تواند گرادیان‌های بزرگی ایجاد کرده و فرآیند یادگیری را مختل کند. در مقابل، در SigLIP، تمرکز بر روی صحت هر زوج به صورت مستقل است. این رویکرد می‌تواند منجر به یادگیری مزهای تصمیم‌گیری پایدارتر شود، زیرا مدل کمتر تحت تاثیر ترکیب خاص نمونه‌های منفی در یک بچ قرار می‌گیرد. تابع زیان سافت‌مکس CLIP شباهت یک زوج مثبت را نسبت به تمام زوج‌های منفی دیگر در همان بچ نرمال می‌کند، در حالی که تابع زیان سیگموئید SigLIP بر روی هر زوج تصویر-متن به طور مستقل عمل می‌کند و نیازی به نرمال‌سازی جهانی در بچ ندارد. این "جداسازی اندازه بچ از تعریف وظیفه" به این معنی است که تاثیر نمونه‌های منفی دیگر در بچ بر روی گرادیان یک زوج خاص، متفاوت از سافت‌مکس است. اگر یک بچ به طور تصادفی حاوی نمونه‌های منفی بسیار گمراه کننده یا نویزی باشد، در سافت‌مکس این نمونه‌ها می‌توانند به شدت بر یادگیری تاثیر بگذارند. در SigLIP، چون هر زوج به طور مستقل ارزیابی می‌شود، تاثیر اینگونه نمونه‌های منفی درون بچ ممکن است کمتر باشد و مدل بتواند تمرکز بیشتری بر روی یادگیری از زوج‌های منفی داشته باشد.

مقاله دوم :

FILIP: Fine-grained Interactive Language-Image Pre-Training

مقاله FILIP یک رویکرد پیش‌آموزش زیان-تصویر تعاملی دقیق‌دانه-Image Pre-training را در مقیاس بزرگ معرفی می‌کند. هدف اصلی این پژوهش، دستیابی به هم‌راستاسازی در سطحی دقیق‌تر و جزئی‌تر بین مؤلفه‌های تصویر و متن است. این مهم از طریق یک سازوکار از حداکثر شباهت توکن-به-توکن (late token interaction) تحقق می‌یابد. این سازوکار از حداکثر شباهت توکن-به-توکن (token-to-token) (cross-modal interaction) بین‌وجهی (between-views) می‌باشد، در سافت‌مکس این نمونه‌ها می‌توانند به شدت بر یادگیری تاثیر بگذارند. در FILIP، توکن‌های بصری (visual tokens) که معمولاً وصله‌های تصویر یا image patches هستند و توکن‌های متنی (words) یا زیرکلمات (subwords) برای هدایت تابع زیان مقابله‌ای بهره می‌برد. با اصلاح تابع زیان مقابله‌ای، موفق می‌شود تا از بیانگری دقیق‌تر موجود بین وصله‌های تصویر و کلمات متنی استفاده کند. هم‌زمان، این مدل قابلیت پیش‌محاسبه آفلاین نمایش‌های (representations) تصویر و متن در زمان استنتاج را حفظ می‌کند، که این ویژگی هم آموزش در مقیاس بزرگ و هم فرآیند استنتاج را کارآمد نگه می‌دارد. نتایج تجربی نشان می‌دهند که FILIP در وظایف پایین‌دستی متعدد زبان-تصویر، از جمله طبقه‌بندی تصویر بدون نمونه و بازیابی تصویر-متن، به عملکرد پیشرفته‌ای (state-of-the-art) دست یافته است.

FILIP از یک سازوکار "حداکثر شباهت توکن-به-توکن" استفاده می‌کند. برای هر توکن بصری شباهت آن با تمام توکن‌های متنی زوج متناظر محاسبه شده و بزرگترین مقدار شباهت انتخاب می‌شود. نتایج نشان می‌دهند که FILIP "به طور قابل توجهی از CLIP در طبقه‌بندی تصویر بدون نمونه بهتر عمل می‌کند... و بهبودهای قابل توجهی در مجموعه داده‌های خاص دامنه مانند Aircrafts حدود ۳۰٪ بهبود) نشان می‌دهد.". علاوه بر این، "تجسم‌سازی هم‌راستایی کلمه-وصله نشان می‌دهد که FILIP قادر به یادگیری ویژگی‌های دقیق‌دانه معنی دار با قابلیت محلی‌سازی امیدوار کننده است". توانایی درک جزئیات و برقراری ارتباط بین بخش‌های خاص یک تصویر و کلمات متناظر با آن‌ها، مدل را در برابر تغییراتی که بر این جزئیات تاثیر می‌گذارند، مقاوم‌تر می‌کند. برخلاف CLIP که عمدهاً به ویژگی‌های سراسری تکیه دارد، FILIP می‌تواند اشیاء را حتی زمانی که در زمینه‌های غیرمعمول قرار دارند یا تنها بخش کوچکی از تصویر را تشکیل می‌دهند، با دقت بیشتری شناسایی کند.

عملکرد قوی در وظایف بدون نمونه (Zero-shot) به این معنی است که مدل می‌تواند مفاهیمی را که در طول آموزش به طور صریح برای آن‌ها برچسب ندیده، شناسایی یا بازیابی کند FILIP. با همراستاسازی دقیق‌دانه، به جای یادگیری یک نمایش سراسری کلی، یاد می‌گیرد که چگونه "اجزای" بصری به "اجزای" متنی مرتبط می‌شوند. این توانایی در تجزیه و ترکیب مفاهیم از طریق اجزای دقیق‌دانه، به مدل اجازه می‌دهد تا ترکیب‌های جدیدی از ویژگی‌ها یا اشیاء را که در داده‌های آموزشی مشاهده نشده‌اند، بهتر درک کند. به عنوان مثال، اگر مدل "گربه راه" و "سگ خالدار" را دیده باشد، توانایی FILIP در همراستاسازی "راه راه" با وصله‌های راه راه و "خالدار" با وصله‌های خالدار، به آن کمک می‌کند تا مفهوم "گربه خالدار" را بدون دیدن نمونه قبلی، بهتر تعمیم دهد، که این یک جنبه از مقاومت در برابر ترکیب‌های جدید و نادیده است.

۲-۲ . پیاده سازی و مقایسه روش های آموزش خصمانه

.۱

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
from torch.utils.data import DataLoader, SubsetRandomSampler

torch.manual_seed(42)
np.random.seed(42)

#CLIP normalization parameters
clip_mean = [0.48145466, 0.4578275, 0.40821073]
clip_std = [0.26862954, 0.26130258, 0.27577711]

#training, validation, and test sets
train_transform = transforms.Compose([
    transforms.Resize((224, 224)), # Resize to 224x224
    transforms.ToTensor(), # Convert to tensor
    transforms.Normalize(mean=clip_mean, std=clip_std) # Normalize
with CLIP values
])

val_test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=clip_mean, std=clip_std)
])

#CIFAR-10 dataset
```

```

train_dataset = torchvision.datasets.CIFAR10(root='./data',
train=True, download=True, transform=None)
test_dataset = torchvision.datasets.CIFAR10(root='./data',
train=False, download=True, transform=None)

#Split
num_train = len(train_dataset)
indices = list(range(num_train))
np.random.shuffle(indices)
train_size = int(0.8 * num_train) # 80% for training, 20% for validation
train_indices, val_indices = indices[:train_size],
indices[train_size:]

train_dataset.transform = train_transform
test_dataset.transform = val_test_transform

val_dataset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=False, transform=val_test_transform)
val_dataset.data = train_dataset.data[val_indices]
val_dataset.targets = [train_dataset.targets[i] for i in val_indices]
train_dataset.data = train_dataset.data[train_indices]
train_dataset.targets = [train_dataset.targets[i] for i in
train_indices]

#labels CIFAR-10
classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
'frog', 'horse', 'ship', 'truck']

#5 random samples
def show_random_samples(dataset, num_samples=5):
    # Create a DataLoader to sample random images
    loader = DataLoader(dataset, batch_size=num_samples,
shuffle=True)
    images, labels = next(iter(loader))

    denorm = transforms.Normalize(
        mean=[-m/s for m, s in zip(clip_mean, clip_std)],
        std=[1/s for s in clip_std]
    )

    fig, axes = plt.subplots(1, num_samples, figsize=(15, 3))
    for i in range(num_samples):
        img = denorm(images[i]).permute(1, 2, 0).numpy() # Convert
        to HWC format for display
        img = np.clip(img, 0, 1) # Ensure pixel values are in [0, 1]
        axes[i].imshow(img)

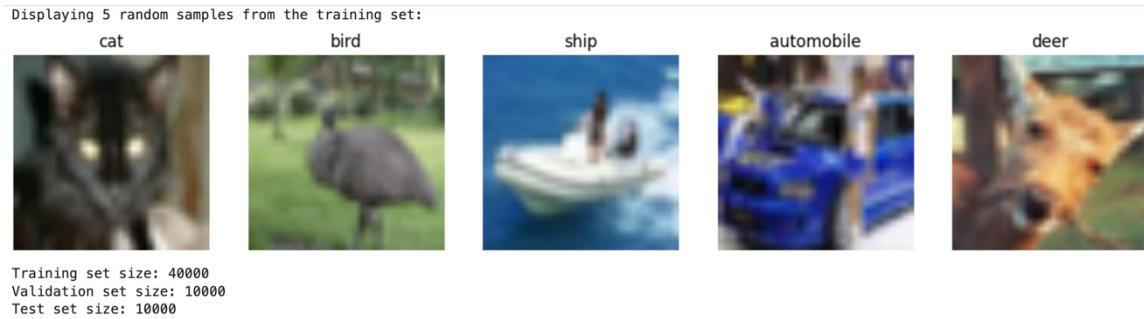
```

```

        axes[i].set_title(classes[labels[i]])
        axes[i].axis('off')
    plt.show()

print("Displaying 5 random samples from the training set:")
show_random_samples(train_dataset)
print(f"Training set size: {len(train_dataset)}")
print(f"Validation set size: {len(val_dataset)}")
print(f"Test set size: {len(test_dataset)}")

```



شكل ١٣.٥ نمونه تصادفي تصوير

.٢

```

import torch
import torchvision
import torchvision.transforms as transforms
from transformers import CLIPProcessor, CLIPModel
import numpy as np

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

#Load CLIP
clip_model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
clip_processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
clip_model = clip_model.to(device)

```

```

clip_model.eval()    # Set CLIP to evaluation mode
print("CLIP model loaded and set to evaluation mode.")

#Load pre-trained ResNet-20
target_model = torch.hub.load("chenyaofu/pytorch-cifar-models",
" cifar10_resnet20", pretrained=True)
target_model = target_model.to(device)
target_model.eval()
print("ResNet-20 model loaded and set to evaluation mode.")

classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
'frog', 'horse', 'ship', 'truck']
text_prompts = [f"a photo of a {cls}" for cls in classes]

text_inputs = clip_processor(text=text_prompts, return_tensors="pt",
padding=True, truncation=True)
text_inputs = {k: v.to(device) for k, v in text_inputs.items()}
with torch.no_grad():
    text_embeddings = clip_model.get_text_features(**text_inputs)
print("Text embeddings for CIFAR-10 classes generated.")
text_embeddings = text_embeddings / text_embeddings.norm(dim=-1,
keepdim=True)
print(f"Text embeddings shape: {text_embeddings.shape}")
torch.save(text_embeddings, "cifar10_text_embeddings.pt")
print("Text embeddings saved to 'cifar10_text_embeddings.pt'.")

```

Text embeddings saved to 'cifar10_text_embeddings.pt'.

.ۯ

```

import torch
import torchvision
import torchvision.transforms as transforms
from transformers import CLIPProcessor, CLIPModel
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader
import torchattacks

torch.manual_seed(42)
np.random.seed(42)

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

```

```

print(f"Using device: {device}")

clip_model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
clip_processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
clip_model = clip_model.to(device)
clip_model.eval()
print("CLIP model loaded and set to evaluation mode.")

target_model = torch.hub.load("chenyaofu/pytorch-cifar-models", "cifar10_resnet20", pretrained=True)
target_model = target_model.to(device)
target_model.eval()
print("ResNet-20 model loaded and set to evaluation mode.")

#CLIP transformations
clip_mean = [0.48145466, 0.4578275, 0.40821073]
clip_std = [0.26862954, 0.26130258, 0.27577711]
test_transform_clip = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=clip_mean, std=clip_std)
])
test_dataset_clip = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=test_transform_clip)
test_loader_clip = DataLoader(test_dataset_clip, batch_size=32, shuffle=False)

test_transform_resnet = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.4914, 0.4822, 0.4465], std=[0.2023, 0.1994, 0.2010])
])
test_dataset_resnet = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=test_transform_resnet)
test_loader_resnet = DataLoader(test_dataset_resnet, batch_size=32, shuffle=False)
text_embeddings = torch.load("cifar10_text_embeddings.pt").to(device)
classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

def evaluate_clip(model, loader, text_embeddings, device):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in loader:

```

```

        images = images.to(device)
        labels = labels.to(device)
        image_features = model.get_image_features(images)
        image_features = image_features /
image_features.norm(dim=-1, keepdim=True)
        logits = image_features @ text_embeddings.t() * 100
        preds = logits.argmax(dim=-1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)
    accuracy = correct / total * 100
    return accuracy

clean_accuracy = evaluate_clip(clip_model, test_loader_clip,
text_embeddings, device)
print(f"CLIP accuracy on clean images: {clean_accuracy:.2f}%")


attack = torchattacks.PGD(target_model, eps=8/255, alpha=2/255,
steps=7)
adversarial_images = []
true_labels = []
for images, labels in test_loader_resnet:
    images = images.to(device)
    labels = labels.to(device)
    adv_images = attack(images, labels)
    adversarial_images.append(adv_images.cpu())
    true_labels.append(labels.cpu())
adversarial_images = torch.cat(adversarial_images)
true_labels = torch.cat(true_labels)

adv_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.Normalize(mean=clip_mean, std=clip_std)
])
adv_images_transformed = torch.stack([adv_transform(img) for img in
adversarial_images])
adv_dataset = torch.utils.data.TensorDataset(adv_images_transformed,
true_labels)
adv_loader = DataLoader(adv_dataset, batch_size=32, shuffle=False)

#Evaluate
adv_accuracy = evaluate_clip(clip_model, adv_loader, text_embeddings,
device)
print(f"CLIP accuracy on adversarial images (PGD attack on ResNet-20): {adv_accuracy:.2f}%")

#visualization
sample_idx = 0
clean_image, label = test_dataset_resnet[sample_idx]

```

```

adv_image = adversarial_images[sample_idx]
noise = adv_image - clean_image

denorm = transforms.Normalize(
    mean=[-m/s for m, s in zip([0.4914, 0.4822, 0.4465], [0.2023,
0.1994, 0.2010])],
    std=[1/s for s in [0.2023, 0.1994, 0.2010]])
)

clean_image = denorm(clean_image).permute(1, 2, 0).numpy()
adv_image = denorm(adv_image).permute(1, 2, 0).numpy()
noise = noise.permute(1, 2, 0).numpy()
clean_image = np.clip(clean_image, 0, 1)
adv_image = np.clip(adv_image, 0, 1)
noise = np.clip(noise + 0.5, 0, 1) # Shift noise for visibility

fig, axes = plt.subplots(1, 3, figsize=(12, 4))
axes[0].imshow(clean_image)
axes[0].set_title(f"Clean Image\nClass: {classes[label]}")
axes[0].axis('off')
axes[1].imshow(adv_image)
axes[1].set_title("Adversarial Image")
axes[1].axis('off')
axes[2].imshow(noise)
axes[2].set_title("Noise (Shifted)")
axes[2].axis('off')
plt.tight_layout()
plt.show()

```

CLIP accuracy on clean images: 87.83%
CLIP accuracy on adversarial images (PGD attack on ResNet-20): 49.87%



.¶

```

import torch
import torchvision
import torchvision.transforms as transforms
from transformers import CLIPProcessor, CLIPModel
from peft import LoraConfig, get_peft_model
import numpy as np
from torch.utils.data import DataLoader, TensorDataset
#!pip install torchattacks
!pip install peft
import torchattacks

# Set random seed for reproducibility
torch.manual_seed(42)
np.random.seed(42)

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Load CLIP model and processor
clip_model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
clip_processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
clip_model = clip_model.to(device)
print("CLIP model loaded.")

# Load pre-trained ResNet-20 model
target_model = torch.hub.load("chenyaofu/pytorch-cifar-models", "cifar10_resnet20", pretrained=True)
target_model = target_model.to(device)
target_model.eval()
print("ResNet-20 model loaded and set to evaluation mode.")

# Load CIFAR-10 test dataset with CLIP transformations
clip_mean = [0.48145466, 0.4578275, 0.40821073]
clip_std = [0.26862954, 0.26130258, 0.27577711]
test_transform_clip = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=clip_mean, std=clip_std)
])
test_dataset_clip = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=test_transform_clip)
test_loader_clip = DataLoader(test_dataset_clip, batch_size=32, shuffle=False)

```

```

# Load CIFAR-10 test dataset for ResNet-20 (32x32, standard
normalization)
test_transform_resnet = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.4914, 0.4822, 0.4465], std=[0.2023,
0.1994, 0.2010])
])
test_dataset_resnet = torchvision.datasets.CIFAR10(root='./data',
train=False, download=True, transform=test_transform_resnet)
test_loader_resnet = DataLoader(test_dataset_resnet, batch_size=32,
shuffle=False)

# Load precomputed text embeddings
text_embeddings = torch.load("cifar10_text_embeddings.pt").to(device)
classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
'frog', 'horse', 'ship', 'truck']

# Apply LoRA to CLIP's vision module
lora_config = LoraConfig(
    r=8,
    lora_alpha=32,
    target_modules=["q_proj", "k_proj", "v_proj", "out_proj"], # Target attention layers in vision transformer
    lora_dropout=0.1,
    bias="none",
    modules_to_save=None
)
clip_model = get_peft_model(clip_model, lora_config)
clip_model = clip_model.to(device)
clip_model.train() # Set to training mode for fine-tuning
print("LoRA applied to CLIP vision module.")

# Generate adversarial training dataset (10,000 images)
attack = torchattacks.PGD(target_model, eps=8/255, alpha=2/255,
steps=7)
adversarial_images = []
true_labels = []
for images, labels in test_loader_resnet:
    images = images.to(device)
    labels = labels.to(device)
    adv_images = attack(images, labels)
    adversarial_images.append(adv_images.cpu())
    true_labels.append(labels.cpu())
adversarial_images = torch.cat(adversarial_images)[:10000] # Limit
to 10,000 samples
true_labels = torch.cat(true_labels)[:10000]

# Transform adversarial images for CLIP

```

```

adv_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.Normalize(mean=clip_mean, std=clip_std)
])
adv_images_transformed = torch.stack([adv_transform(img) for img in
adversarial_images])[:10000]
adv_dataset = TensorDataset(adv_images_transformed, true_labels)
adv_loader = DataLoader(adv_dataset, batch_size=32, shuffle=True)

# Define optimizer for LoRA parameters
optimizer = torch.optim.AdamW(clip_model.parameters(), lr=1e-4)

# Fine-tune CLIP for one epoch on adversarial images
print("Starting adversarial fine-tuning with LoRA...")
clip_model.train()
for images, labels in adv_loader:
    images = images.to(device)
    labels = labels.to(device)

    optimizer.zero_grad()
    # Compute image embeddings
    image_features = clip_model.get_image_features(images)
    image_features = image_features / image_features.norm(dim=-1,
keepdim=True)
    # Compute logits
    logits = image_features @ text_embeddings.t() * 100 # Scale for
stability
    # Compute loss
    loss = torch.nn.CrossEntropyLoss()(logits, labels)
    # Backpropagate
    loss.backward()
    optimizer.step()

print(f"Fine-tuning completed. Final loss: {loss.item():.4f}")

# Function to evaluate CLIP (zero-shot classification)
def evaluate_clip(model, loader, text_embeddings, device):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in loader:
            images = images.to(device)
            labels = labels.to(device)
            image_features = model.get_image_features(images)
            image_features = image_features /
image_features.norm(dim=-1, keepdim=True)
            logits = image_features @ text_embeddings.t() * 100
            preds = logits.argmax(dim=-1)

```

```

        correct += (preds == labels).sum().item()
        total += labels.size(0)
    accuracy = correct / total * 100
    return accuracy

# Evaluate CLIP on clean test set after fine-tuning
clip_model.eval()
clean_accuracy = evaluate_clip(clip_model, test_loader_clip,
text_embeddings, device)
print(f"CLIP accuracy on clean images after fine-tuning:
{clean_accuracy:.2f}%")


# Generate new adversarial test set for evaluation
adversarial_images_test = []
true_labels_test = []
for images, labels in test_loader_resnet:
    images = images.to(device)
    labels = labels.to(device)
    adv_images = attack(images, labels)
    adversarial_images_test.append(adv_images.cpu())
    true_labels_test.append(labels.cpu())
adversarial_images_test = torch.cat(adversarial_images_test)
true_labels_test = torch.cat(true_labels_test)

# Transform adversarial test images for CLIP
adv_images_test_transformed = torch.stack([adv_transform(img) for img
in adversarial_images_test])
adv_test_dataset = TensorDataset(adv_images_test_transformed,
true_labels_test)
adv_test_loader = DataLoader(adv_test_dataset, batch_size=32,
shuffle=False)

# Evaluate CLIP on adversarial test set after fine-tuning
adv_accuracy = evaluate_clip(clip_model, adv_test_loader,
text_embeddings, device)
print(f"CLIP accuracy on adversarial images after fine-tuning:
{adv_accuracy:.2f}%")

```

Fine-tuning completed. Final loss: 0.5233

CLIP accuracy on clean images after fine-tuning: 92.52%

CLIP accuracy on adversarial images after fine-tuning: 86.66%

.Δ

```

import torch
import torchvision
import torchvision.transforms as transforms
from transformers import CLIPProcessor, CLIPModel, CLIPVisionModel
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader, Subset
import torchattacks
import torch.nn.functional as F
import torch.optim as optim
import torch.nn as nn

# Set random seed for reproducibility
torch.manual_seed(42)
np.random.seed(42)

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Load CLIP model and processor
clip_model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
clip_processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
clip_model = clip_model.to(device)
clip_model.eval()
print("CLIP model loaded and set to evaluation mode.")

class CLIPWrapper(nn.Module):
    def __init__(self, clip_model):
        super().__init__()
        self.clip_model = clip_model

    def forward(self, x):
        return self.clip_model.get_image_features(x)

# Load CIFAR-10 datasets
clip_mean = [0.48145466, 0.4578275, 0.40821073]
clip_std = [0.26862954, 0.26130258, 0.27577711]
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=clip_mean, std=clip_std)
])
test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])

```

```

        transforms.Normalize(mean=clip_mean, std=clip_std)
    ])
train_dataset = torchvision.datasets.CIFAR10(root='./data',
train=True, download=True, transform=train_transform)
test_dataset = torchvision.datasets.CIFAR10(root='./data',
train=False, download=True, transform=test_transform)

# Subset training data to 10,000 samples
indices = np.random.choice(len(train_dataset), 10000, replace=False)
train_subset = Subset(train_dataset, indices)
train_loader = DataLoader(train_subset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Load precomputed text embeddings
text_embeddings = torch.load("cifar10_text_embeddings.pt").to(device)
classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
'frog', 'horse', 'ship', 'truck']

# TeCoA contrastive loss function
def contrastive_loss(image_features, text_features, labels,
tau=0.07):
    image_features = image_features / image_features.norm(dim=-1,
keepdim=True)
    text_features = text_features / text_features.norm(dim=-1,
keepdim=True)
    logits = (image_features @ text_features.t()) / tau
    loss = F.cross_entropy(logits, labels)
    return loss

# PGD attack for contrastive loss
def pgd_attack(model, images, text_features, labels, eps=1/255,
alpha=1/255, steps=2):
    images = images.clone().detach().requires_grad_(True)
    orig_images = images.clone().detach()

    for _ in range(steps):
        model.zero_grad()
        image_features = model.get_image_features(images)
        loss = contrastive_loss(image_features, text_features,
labels)
        loss.backward()
        with torch.no_grad():
            adv_images = images + alpha * images.grad.sign()
            delta = torch.clamp(adv_images - orig_images, min=-eps,
max=eps)
            images = torch.clamp(orig_images + delta, min=0,
max=1).requires_grad_(True)
    return images

```

```

# Visual Prompt Tuning (VPT) module
class VisualPrompt(nn.Module):
    def __init__(self, prompt_size=5, embed_dim=512):
        super().__init__()
        self.prompt = nn.Parameter(torch.randn(prompt_size,
embed_dim))

    def forward(self, x):
        batch_size = x.size(0)
        prompt = self.prompt.unsqueeze(0).repeat(batch_size, 1, 1)
        return torch.cat([prompt, x], dim=1)

# Evaluation function
def evaluate_clip(model, loader, text_embeddings, device, vpt=None):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in loader:
            images = images.to(device)
            labels = labels.to(device)
            if vpt is not None:
                image_features =
model.vision_model(images).last_hidden_state
                image_features = vpt(image_features)
                image_features =
model.vision_model_ln_post(image_features)
                image_features = image_features[:, 0, :] # CLS token
            else:
                image_features = model.get_image_features(images)
                image_features = image_features /
image_features.norm(dim=-1, keepdim=True)
                logits = image_features @ text_embeddings.t() * 100
                preds = logits.argmax(dim=-1)
                correct += (preds == labels).sum().item()
                total += labels.size(0)
            accuracy = correct / total * 100
    return accuracy

# TeCoA training function
def train_tecoa(model, train_loader, text_embeddings, device,
epochs=3, vpt=None, finetune=False):
    if vpt is not None:
        optimizer = optim.SGD(vpt.parameters(), lr=40, momentum=0.9)
        print("Training with Visual Prompt Tuning")
    else:

```

```

        optimizer = optim.SGD(model.parameters(), lr=1e-5,
momentum=0.9)
        print("Training with Finetuning")

model.train()
for epoch in range(epochs):
    total_loss = 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        adv_images = pgd_attack(model, images, text_embeddings,
labels)
        model.zero_grad()
        if vpt is not None:
            image_features =
model.vision_model(adv_images).last_hidden_state
            image_features = vpt(image_features)
            image_features =
model.vision_model.ln_post(image_features)
            image_features = image_features[:, 0, :] # CLS token
        else:
            image_features = model.get_image_features(adv_images)
        loss = contrastive_loss(image_features, text_embeddings,
labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        print(f"Epoch {epoch+1}/{epochs}, Loss:
{total_loss/len(train_loader):.4f}")

# Train with Finetuning
print("Starting TeCoA Finetuning")
train_tecoa(clip_model, train_loader, text_embeddings, device,
finetune=True)

# Evaluate on clean test set
clean_accuracy_ft = evaluate_clip(clip_model, test_loader,
text_embeddings, device)
print(f"TeCoA Finetuning - Clean accuracy: {clean_accuracy_ft:.2f}%")

# Generate adversarial test images (100-step PGD)
clip_wrapper = CLIPWrapper(clip_model).to(device)
attack = torchattacks.PGD(clip_wrapper, eps=1/255, alpha=1/255,
steps=100)
attack.set_device(device) # Explicitly set device
adversarial_images = []
true_labels = []
for images, labels in test_loader:
    images = images.to(device)

```

```

    labels = labels.to(device)
    adv_images = attack(images, labels)
    adversarial_images.append(adv_images.cpu())
    true_labels.append(labels.cpu())
adversarial_images = torch.cat(adversarial_images)
true_labels = torch.cat(true_labels)

# Create adversarial dataset
adv_dataset = torch.utils.data.TensorDataset(adversarial_images,
true_labels)
adv_loader = DataLoader(adv_dataset, batch_size=32, shuffle=False)

adv_accuracy_ft = evaluate_clip(clip_model, adv_loader,
text_embeddings, device)
print(f"TeCoA Finetuning - Adversarial accuracy (100-step PGD):
{adv_accuracy_ft:.2f}%")


# Train with Visual Prompt Tuning
print("\nStarting TeCoA Visual Prompt Tuning")
vpt = VisualPrompt(prompt_size=5, embed_dim=512).to(device)
train_tecoa(clip_model, train_loader, text_embeddings, device,
vpt=vpt, epochs=10)

# Evaluate with VPT
clean_accuracy_vpt = evaluate_clip(clip_model, test_loader,
text_embeddings, device, vpt=vpt)
adv_accuracy_vpt = evaluate_clip(clip_model, adv_loader,
text_embeddings, device, vpt=vpt)
print(f"TeCoA VPT - Clean accuracy: {clean_accuracy_vpt:.2f}%")
print(f"TeCoA VPT - Adversarial accuracy (100-step PGD):
{adv_accuracy_vpt:.2f}%")


sample_idx = 0
clean_image, label = test_dataset[sample_idx]
adv_image = adversarial_images[sample_idx]
noise = adv_image - clean_image

denorm = transforms.Normalize(
    mean=[-m/s for m, s in zip(clip_mean, clip_std)],
    std=[1/s for s in clip_std]
)
clean_image = denorm(clean_image).permute(1, 2, 0).numpy()
adv_image = denorm(adv_image).permute(1, 2, 0).numpy()
noise = noise.permute(1, 2, 0).numpy()

clean_image = np.clip(clean_image, 0, 1)
adv_image = np.clip(adv_image, 0, 1)
noise = np.clip(noise + 0.5, 0, 1) # Shift for visibility

```

```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
axes[0].imshow(clean_image)
axes[0].set_title(f"Clean Image\nClass: {classes[label]}")
axes[0].axis('off')
axes[1].imshow(adv_image)
axes[1].set_title("Adversarial Image")
axes[1].axis('off')
axes[2].imshow(noise)
axes[2].set_title("Noise (Shifted)")
axes[2].axis('off')
plt.tight_layout()
plt.show()
```

CLIP model loaded and set to evaluation mode.

Starting TeCoA Finetuning

Training with Finetuning

Epoch 1/10, Loss: 1.9146

Epoch 2/10, Loss: 1.4189

Epoch 3/10, Loss: 1.1511

Epoch 4/10, Loss: 0.9888

Epoch 5/10, Loss: 0.8730

Epoch 6/10, Loss: 0.7874

Epoch 7/10, Loss: 0.7102

Epoch 8/10, Loss: 0.6434

Epoch 9/10, Loss: 0.5871

Epoch 10/10, Loss: 0.5401

TeCoA Finetuning - Clean accuracy: 86.13%

TeCoA Finetuning - Adversarial accuracy (100-step PGD): 85.39%