به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین امتیازی

پرسش ۱	نام و نام خانوادگی	آرمین قاسمی
	شماره دانشجویی	۸۱۰۱۰۰۱۹۸
پرسش ۲	نام و نام خانوادگی	امیرحسین ثمودی
	شماره دانشجویی	۸۱۰۱۰۰۱۰۸
	مهلت ارسال پاسخ	۱۴۰۳.۱۲.۲۷

فهرست

شکل‌ها

جدول‌ها

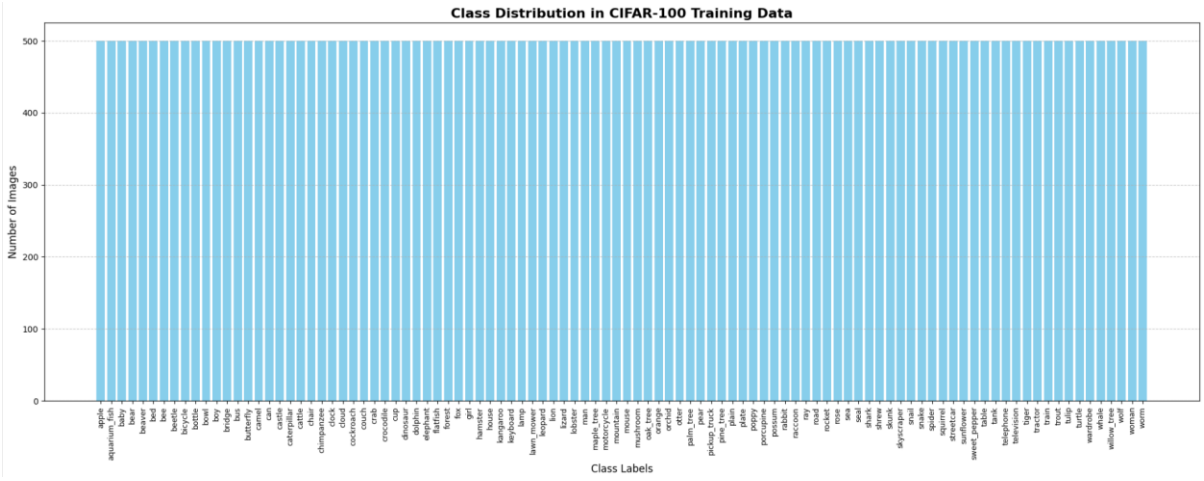
پرسش ۱ – تحلیل عملکرد شبکه‌های عصبی تحت حملات متخاصم

۱-۱: آموزش مدل ResNet روی تصاویر نویزی

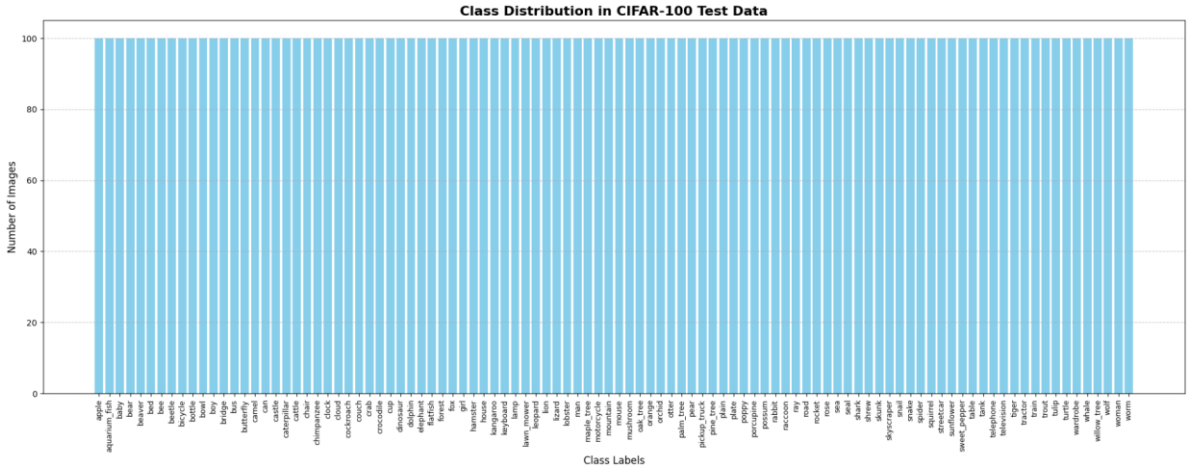
در این بخش مدل ResNet18 را بدون استفاده از وزن‌های پیش آموزش دیده استفاده می‌کنیم. دلیل ترجیح این مدل به ResNet34 ،سایز کوچک‌تر آن هست (تعداد پارامترهای ResNet18 حدود ۱۲ میلیون می‌باشد در مقایسه با ۲۲ میلیون پارامتر مدل ResNet34) که با توجه به تسک ما که طبقه‌بندی دیتاست CIFAR-100 هست و تعداد نمونه‌های آموزشی زیادی در اختیار نداریم، احتمال بیش‌برازش (Overfitting) مدل را کاهش داده و همچنین آموزش سریع‌تری نیز خواهد داشت.

۱: بارگذاری دیتاست CIFAR-100

در مرحله اول دیتاست مورد نظر را بارگذاری کرده و از نظر آماری تحلیل می‌کنیم. دیتاست CIFAR-100 شامل ۵۰۰۰۰ نمونه آموزشی و ۱۰۰۰۰ نمونه تست می‌باشد.



شکل ۱: نمودار توزیع کلاس‌ها در مجموعه آموزش



شکل ۲: نمودار توزیع کلاس‌ها در مجموعه تست

همانطور که می‌بینیم توزیع نمونه‌های همه کلاس‌ها یکنواخت می‌باشد که به آموزش بهتر مدل کمک می‌کند و نیازی به استفاده از روش های داده افزایی برای بالانس کردن کلاس‌ها نیست.

برای آموزش بهتر مدل، هنگام بارگذاری تصاویر، آن ها را با مقادیر میانگین و انحراف معیار این دیتاست نرمال می‌کنیم تا میانگین تصاویر صفر و واریانس آن‌ها یک شود.

۲: افزودن نویز گوسی

در این مرحله طبق خواسته سوال یک نویز گوسی با میانگین صفر و واریانس ۰.۰۵ به داده‌های آموزشی اضافه می‌کنیم.

```
transform_noisy = transforms.Compose([
    transforms.ToTensor(),
    AddGaussianNoise(mean=0., std=noise_std),
    transforms.Normalize(mean, std)
])

trainset_noisy = torchvision.datasets.CIFAR100(
    root=base_path, train=True, download=True, transform=transform_noisy
)
```

همانطور که میبینیم بعد از افزودن نویز، دوباره تصاویر را نرمالایز میکنیم تا آموزش مدل روی آن‌ها بهتر صورت گیرد.

۳: آموزش مدل ResNet روی داده‌های نویزی و بدون نویز

حال میخواهیم مدل ResNet18 را یک بار برروی داده های اصلی و بار دیگر برروی داده های نویزی آموزش دهیم و عملکرد آن ها را با هم مقایسه کنیم.

در ابتدا لازم است تا بخشی از مجموعه دادگان آموزش را به عنوان داده‌های اعتبارسنجی (Validation) کنار بگذاریم. برای اینکار ۱۰ درصد دادگان آموزشی (۵ هزار نمونه) را به صورت رندوم انتخاب کرده و جداسازی کرده و مدل را روی ۴۵ هزار داده باقی‌مانده آموزش می‌دهیم.

سایز بچ را برای داده‌های آموزشی، اعتبارسنجی و تست برابر ۲۵۶ درنظر می‌گیریم

قبل از شروع لازم است تغییراتی در ساختار مدل ResNet18 بدهیم تا با مسئله ما بهتر سازگار شود. اول از همه لازم است سایز لایه Fully-Connected انتهایی که برای طبقه بندی استفاده میشود را به ۱۰۰ تغییر دهیم، زیرا دیتاست مورد نظر ما ۱۰۰ کلاس دارد. نکته دیگر اینکه مدل ResNet برای دیتاست ImageNet با سایز تصاویر ۲۲۴ در ۲۲۴ ساخته شده است، ولی دیتاست ما در اینجا سایز بسیارکوچکتری

دارد (۳۲ در ۳۲). با بررسی ساختار مدل مطابق شکل زیر میبینیم که لایه کانولوشن اول این مدل با Stride برابر ۲، و سپس لایه Maxpooling ابعاد Feature Map ورودی را در همان ابتدای کار یک چهارم می‌کنند که برای سایز تصویر ورودی کوچکی که داریم عملا باعث از دست رفتن بخش زیادی از داده های تصویر ورودی می‌شود.

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 16, 16]	9,408
BatchNorm2d-2	[-1, 64, 16, 16]	128
ReLU-3	[-1, 64, 16, 16]	0
MaxPool2d-4	[-1, 64, 8, 8]	0

شکل ۳: اولین لایه کانولوشن مدل **ResNet18**

بنابراین برای عملکرد بهتر مدل روی دیتاست مورد نظر، لایه کانولوشن اولیه را تغییر داده تا سایز را کاهش ندهد و همچنین لایه MaxPoling را نیز با یک لایه Identity جایگزین میکنیم.

```
def create_resnet18_cifar():

    model = models.resnet18(weights=None)
    # Original: nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,
bias=False)
    model.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1,
bias=False)
    model.maxpool = nn.Identity()

    num_ftrs = model.fc.in_features
    model.fc = nn.Linear(num_ftrs, 100)

    return model.to(device)
```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 32, 32]	1,728
BatchNorm2d-2	[-1, 64, 32, 32]	128
ReLU-3	[-1, 64, 32, 32]	0
Identity-4	[-1, 64, 32, 32]	0
Conv2d-5	[-1, 64, 32, 32]	36,864

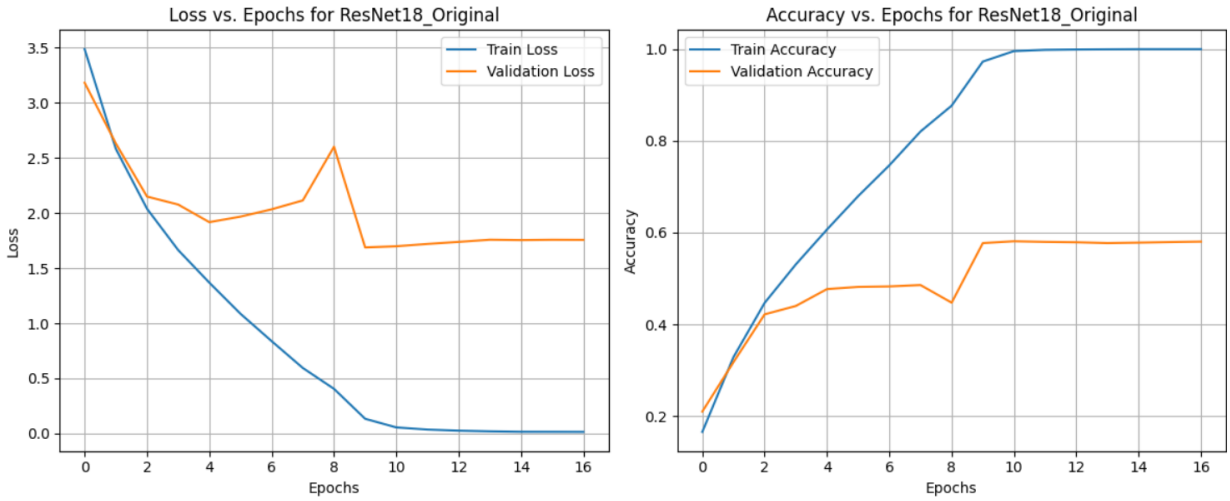
شکل ۴: اولین لایه مدل **ResNet18** تغییر یافته

حال هایپرپارامترهای آموزش را مشخص می‌کنیم. تعداد ایپاک ها را مطابق خواسته سوال ۲۰ قرار میدهیم. نرخ یادگیری را ۰.۰۰۱ قرار داده و همچنین متد Variable Learning Rate را هم برای عملکرد

بهتر مدل اعمال میکنیم. همچنین Weight Decay با مقدار 1e-4 و Early Stop با Patience = 7 را نیز اعمال میکنیم. بهترین مدل بدست آمده را نیز ذخیره میکنیم تا بعدا از آن استفاده کنیم.

از اپتیمایزر Adam و تابع هزینه CrossEntropy استفاده می‌نماییم.

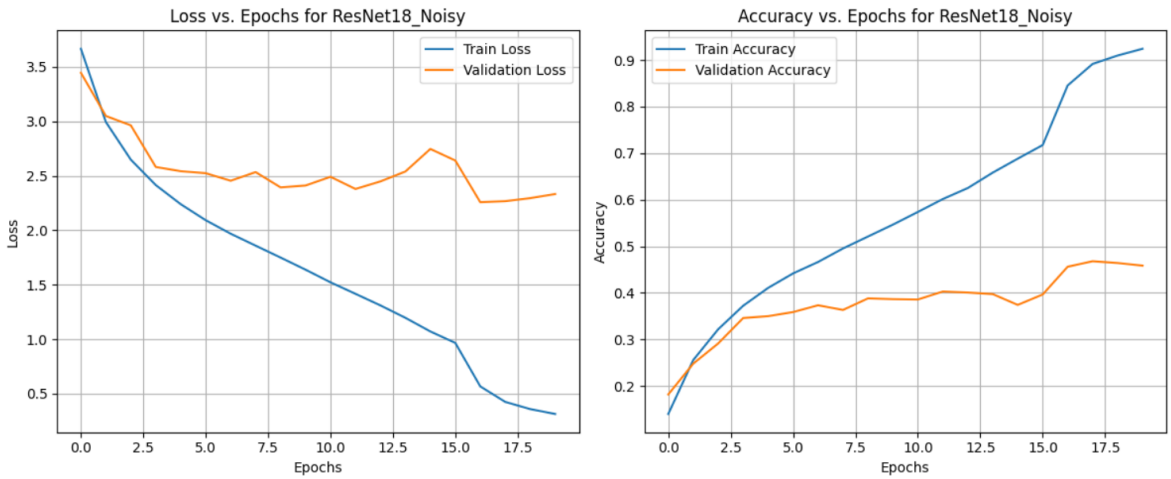
۴: نمودارهای دقت و خطای اعتبارسنجی



شکل ۵: نمودار دقت و خطای آموزش و اعتبارسنجی در آموزش برروی داده های اصلی

دقت مدل در این حالت (آموزش برروی داده های اصلی) به ۶۰.۲ درصد می‌رسد. (ارزیابی شده برروی داده های تست)

حال مدل دیگری را با همین تنظیمات اینبار برروی داده های آموزشی نویزی آموزش میدهیم.



شکل ۶: نمودار دقت و خطای آموزش و اعتبارسنجی در آموزش برروی داده های نویزی

دقت مدل در این حالت (آموزش برروی داده های اصلی) به ۳۶.۴ درصد می‌رسد.

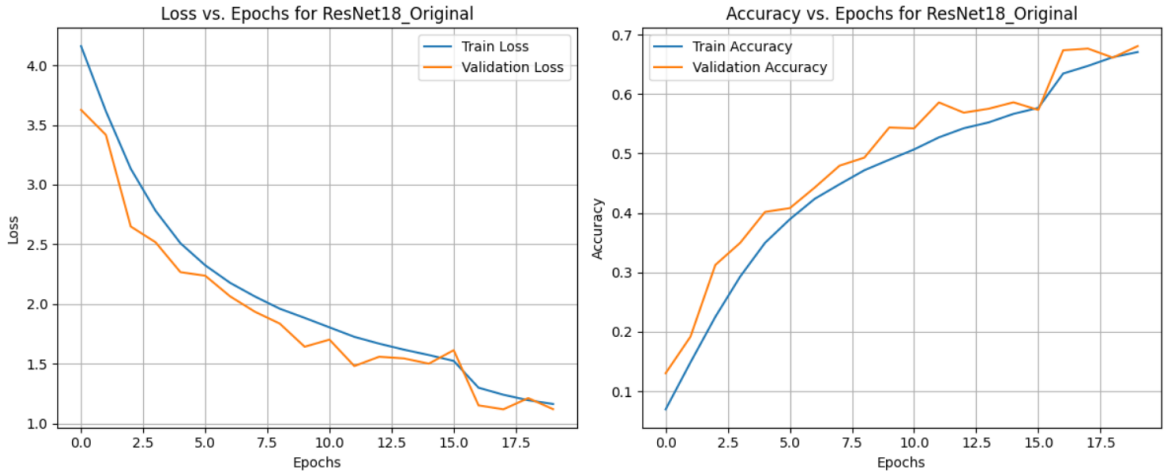
همانطور که در شکل های ۵ و ۶ مشخص است، به وضوح مدل دچار بیش برازش شده و دقت برروی دادگان آموزشی به نزدیک ۱۰۰ درصد رسیده، حال آنکه عملکرد برروی دادگان اعتبارسنجی بعد از گذشت چند ایپاک دیگر دچار تغییر محسوسی نشده است. با توجه به سایز بزرگ مدل در مقایسه با دیتاست، میشد این اتفاق را پیشبینی نیز نمود.

برای رفع این مشکل، دادگان آموزشی را در هر دو حالت با اعمال Data Augmentation تغییر می‌دهیم تا در هر ایپاک تصاویر کمی متفاوتی به مدل اعمال شود و مدل را وادار کند تا ویژگی های واقعی را از تصاویر استخراج کند.

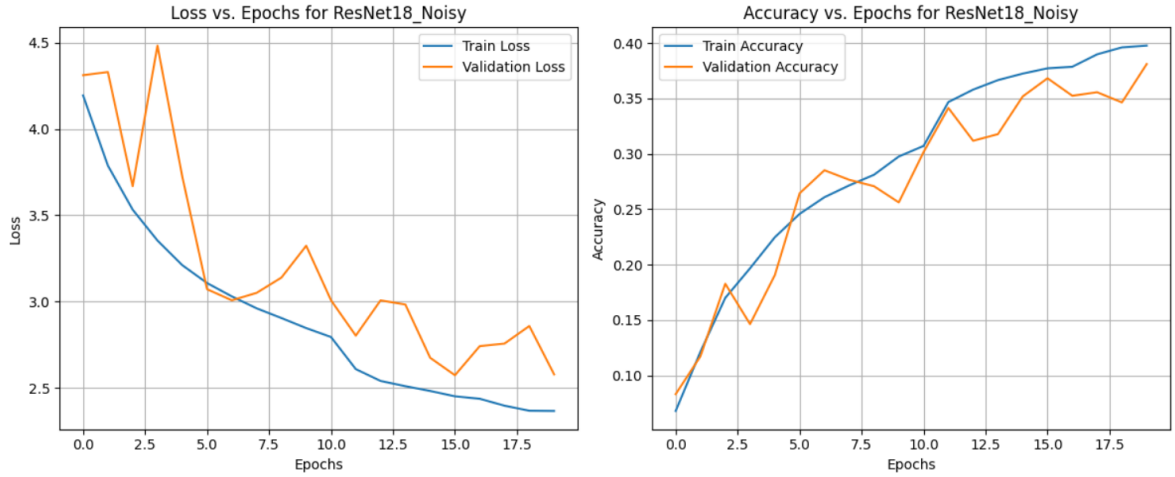
```
transform_train_augmented = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.AutoAugment(policy=transforms.AutoAugmentPolicy.CIFAR10),
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])

transform_noisy = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.AutoAugment(policy=transforms.AutoAugmentPolicy.CIFAR10),
    transforms.ToTensor(),
    AddGaussianNoise(mean=0., std=noise_std),
    transforms.Normalize(mean, std)
])
```

از AutoAugmentation یی که برروی دادگان CIFAR-10 تعریف شده است استفاده کرده ایم، زیرا شباهت زیادی به دادگان آموزش ما دارند.

حال مدل ها را بار دیگر آموزش میدهیم (با همان تنظیمات قبلی).



شکل ۷: نمودار دقت و خطای آموزش و اعتبارسنجی در آموزش برروی داده های اصلی (بعد از داده‌افزایی)



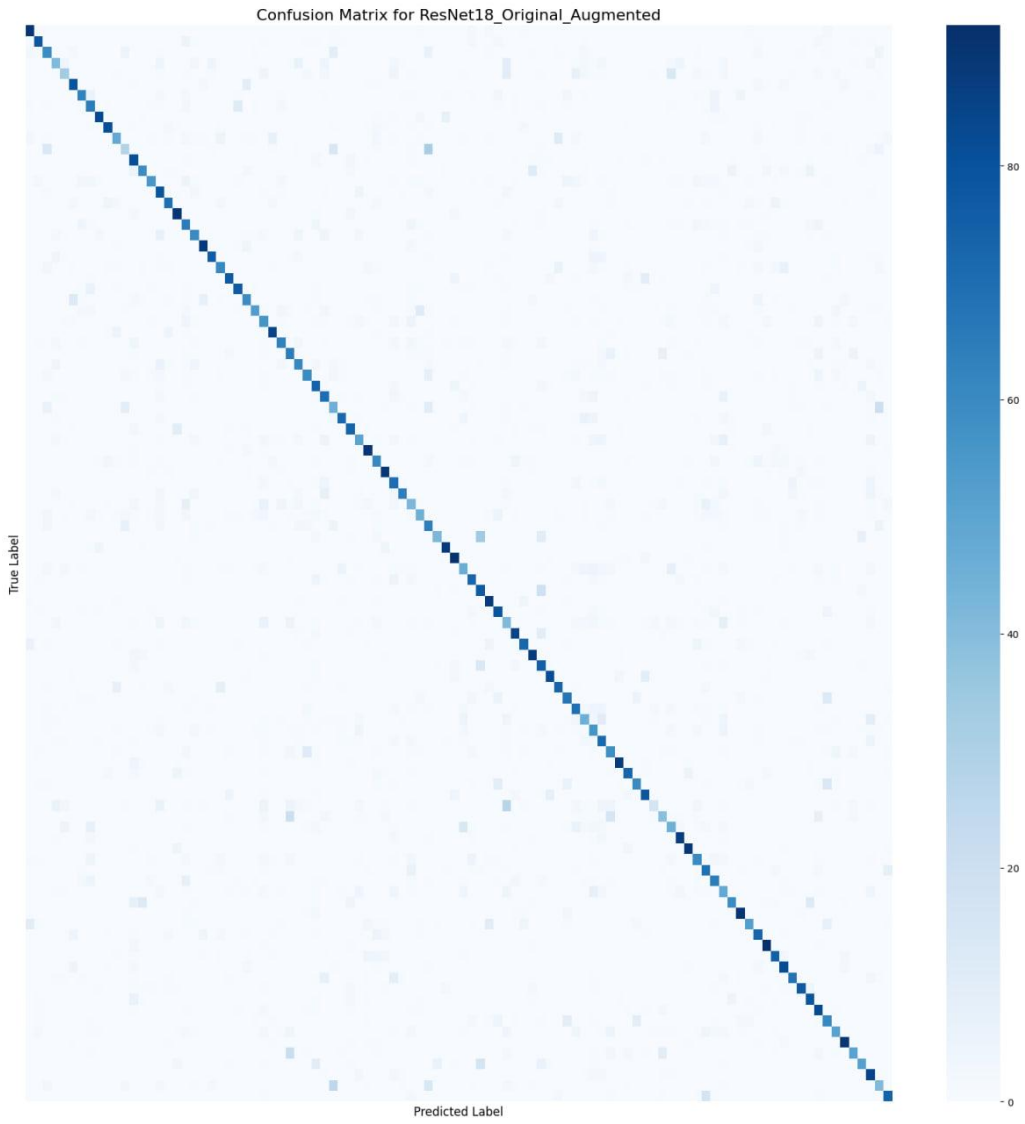
شکل ۸: نمودار دقت و خطای آموزش و اعتبارسنجی در آموزش برروی داده های نویزی (بعد از داده‌افزایی)

به وضوح مشخص است که نمودارهای دقت و خطای آموزش در هر دو حالت تغییرات Smooth تری دارند و با طی کردن ایپاک های بیشتر دقت مدل برروی دادگان آموزش و اعتبارسنجی به صورت همگام با هم بهبود یافته است (مدل اشباع نشده و دچار بیش‌برازش نمیشود) و احتمالا با آموزش در ایپاک های بیشتر به دقت های بالاتری نیز میشد دست یافت. همچنین مدل ها به دقت نهایی بیشتری نیز رسیده اند.

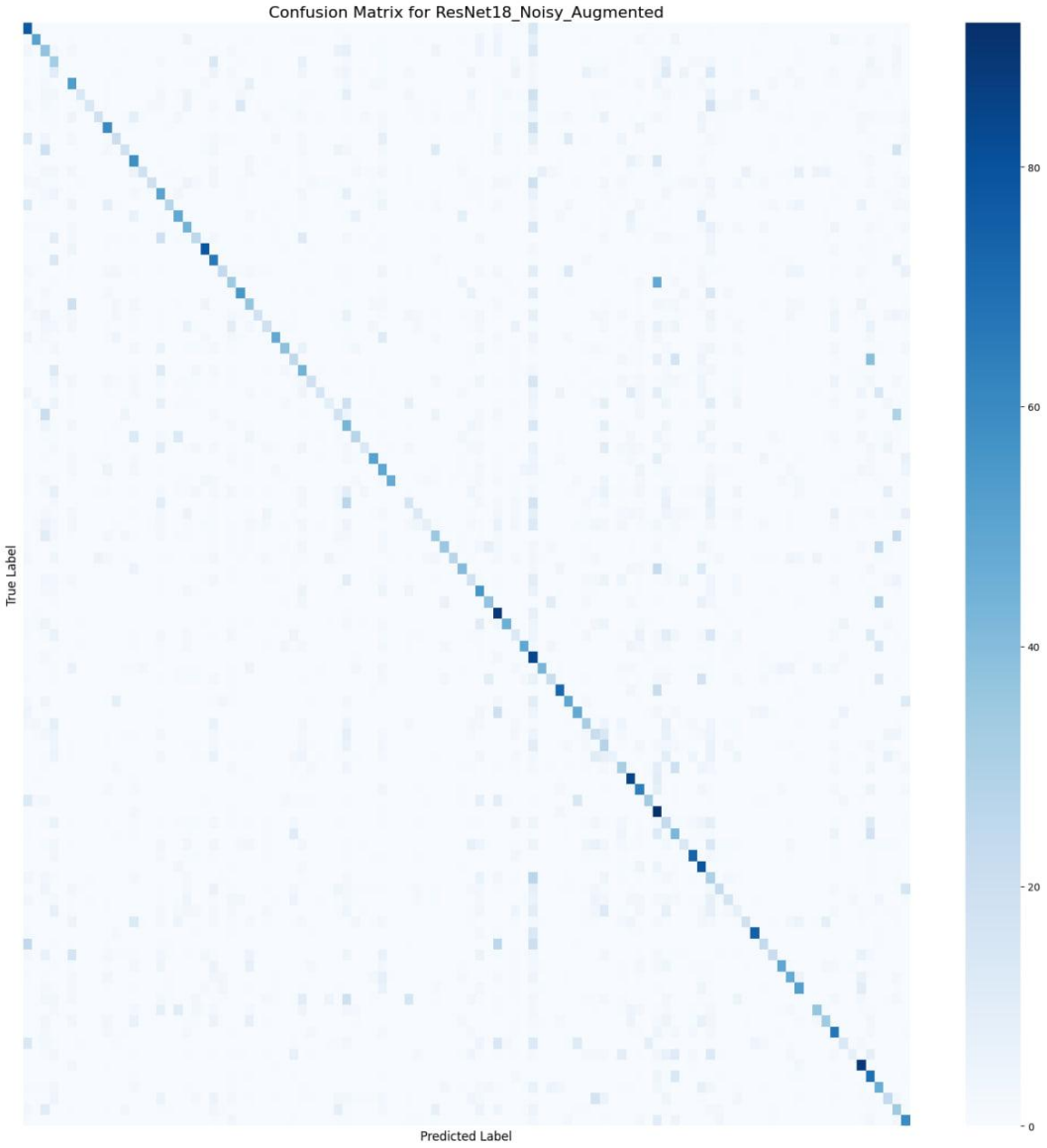
۵: مقایسه عملکرد مدل ها و تحلیل اثر نویز

بعد از اعمال DataAugmentation، دقت مدل در آموزش برروی داده های اصلی به ۶۷.۱۴ درصد می‌رسد و در آموزش برروی داده‌های نویزی به دقت ۳۷.۰۴ درصد (برروی داده های تست) می‌رسد که نشان دهنده کاهش شدید عملکرد مدل به تقریبا نصف حالت اولیه در آموزش با داده های نویز می‌باشد.

برای مقایسه بهتر ماتریس های آشفتگی را در هر دو حالت رسم میکنیم.



شکل ۹: ماتریس آشفتگی مدل آموزش داده شده با داده های اصلی

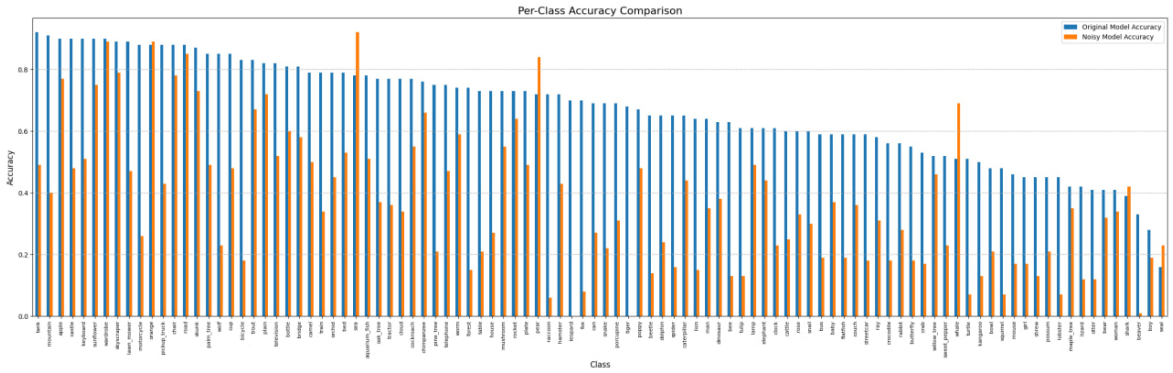


شکل ۱۰: ماتریس آشفتگی مدل آموزش داده شده با داده های نویزی

همانطور که در شکل ۹ می‌بینیم مدل اصلی عملکرد قابل قبولی برروی اکثر کلاس ها دارد و اشتباهات یک کلاس معمولا متمرکز بر یک یا دو کلاس اشتباه هستند که میتواند ناشی از آموزش ناکافی مدل و نزدیک بودن ویژگی های این کلاس ها بهم دیگر باشد.

ولی در شکل ۱۰ می‌بینیم که با افزودن نویز به داده های آموزشی، کلاس های پیشبینی شده پراکندگی بسیار زیادی دارند که با ماهیت تصادفی نویز نیست سازگار است. یک نکته جالب دیگر اینکه تراکم کلاس های پیشبینی شده در برخی از کلاس های میانی به طرز قابل توجهی زیاد است! این امر را میتوان با توزیع گوسی نویز به این صورت توجیه کرد که نویز اضافه شده به اغلب تصاویر مشابه یکدیگرند (در وسط توزیع قرار دارند) که این نویز مشابه باعث شده مدل در تشخیص کلاس های مختلف، آن ها را به اشتباه به یک کلاس مشخصی متناظر کند. (بیشترین کلاس پیشبینی شده گلابی است!)

برای بررسی بهتر تاثیر نویز، دقت مدل به تفکیک کلاس را در هر دو حالت محاسبه و رسم می‌کنیم.



شکل ۱۱: دقت مدل ها به تفکیک کلاس

نکته جالب در این نمودار تغییرات رندوم دقت مدل در کلاس های مختلف است که با ماهیت تصادفی نویز نیز سازگاری دارد. به عبارتی تاثیری که نویز در دقت هر کلاس میگذارد میتواند به هر صورتی باشد و امری قابل پیشبینی یا قابل تحلیل نیست.

همچنین برای تحلیل بیشتر دقت مدل ها را بر Super-Class های دیتاست CIFAR-100 نیز محاسبه میکنیم (این دیتاست از ۲۰ سوپر-کلاس تشکیل شده که کلاس های مشابه همدیگر را در یک کلاس بزرگتر قرار میدهد.)

دقت مدل اصلی در پیشبینی سوپر-کلاس ها ۷۸.۸۶ درصد و دقت مدل نویزی ۵۰.۴۱ درصد می‌باشد. همانطور که میبینیم مدل نویزی در پیشبینی سوپر کلاس ها نیز حدود ۳۰ درصد افت دقت دارد که مشابه افت دقت مدل روی کلاس های اصلی است. بنابراین نتیجه میگیریم که نویز اضافه شده تاثیر مخربی دارد و عملکرد مدل را به حدی تخریب میکند که کلاس اشتباهی که پیشبینی میکند نزدیک به کلاس اصلی هم نیست (که باز هم نشان دهنده تاثیر مخرب و همچنین تصادفی نویز بر دقت مدل می‌باشد).

۲-۱: انتقال یادگیری با ViT روی Flowers-102

۱: بارگذاری مدل

ابتدا مدل ترنسفورمر vit_base_patch16_224 را با وزن های پیش آموزش دیده بارگذاری کرده و سایز لایه خروجی آن را به ۱۰۲ تغییر میدهیم.

```
ViTForImageClassification(
  (vit): ViTModel(
    (embeddings): ViTEmbeddings(
      (patch_embeddings): ViTPatchEmbeddings(
        (projection): Conv2d(3, 768, kernel_size=(16, 16), stride=(16,
16))
      )
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (encoder): ViTEncoder(
      (layer): ModuleList(
        (0-11): 12 x ViTLayer(
          (attention): ViTAttention(
            (attention): ViTSelfAttention(
              (query): Linear(in_features=768, out_features=768,
bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768,
bias=True)
            )
            (output): ViTSelfOutput(
              (dense): Linear(in_features=768, out_features=768,
bias=True)
              (dropout): Dropout(p=0.0, inplace=False)
            )
          )
          (intermediate): ViTIntermediate(
            (dense): Linear(in_features=768, out_features=3072,
bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): ViTOutput(
            (dense): Linear(in_features=3072, out_features=768,
bias=True)
            (dropout): Dropout(p=0.0, inplace=False)
          )
          (layernorm_before): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
          (layernorm_after): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        )
      )
    )
    (layernorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  )
  (classifier): Linear(in_features=768, out_features=102, bias=True)
)
```

ساختار مدل به صورت بالا است و تعداد پارامترهای آن ۸۵ میلیون می‌باشد که عدد بزرگی است.

همچنین این مدل یک Processor دارد که وظیفه اش آماده کردن تصاویر ورودی برای اعمال به مدل است. (مانند تطبیق سایز به ۲۲۴ و نرمال سازی تصاویر و ...)

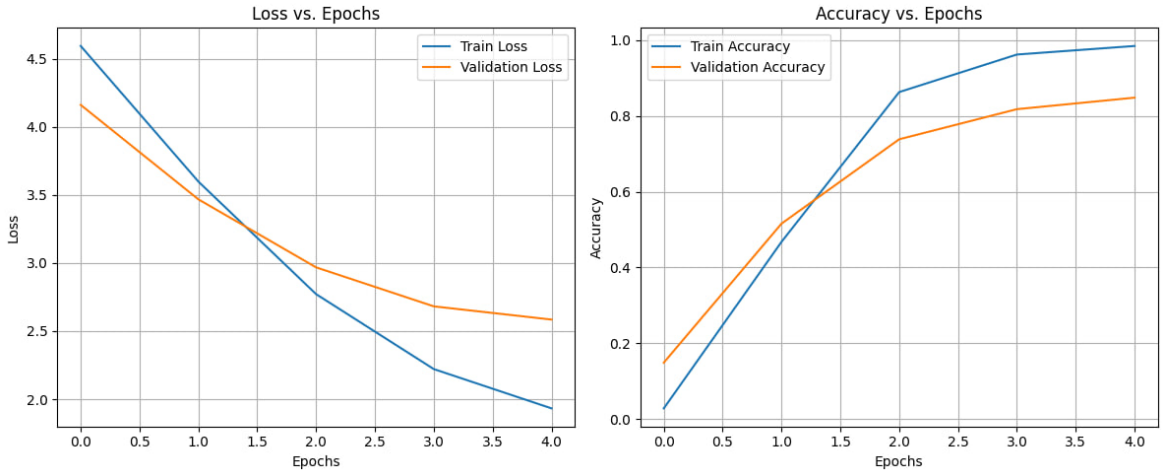
سپس دیتاست Flowers-102 را بارگذاری میکنیم. این دیتاست دارای ۱۰۲۰ تصویر آموزشی، ۱۰۲۰ تصویر اعتبارسنجی و ۶۱۴۹ تصویر تست است که توزیع عجیبی به نظر می‌رسد! همچنین تعداد داده های آموزشی به وضوح کم است و برای آموزش مدل ترنسفور بزرگی که داریم مناسب به نظر نمی‌رسد.

۲: تنظیم مجدد (Fine-tune) مدل روی Flowers-102

در این بخش مدل ViT با وزن های از پیش تمرین داده شده را روی دیتاستمان برای ۵ ایپاک آموزش میدهیم تا تنظیم شود.

بچ سایز را برای همه مجموعه های داده ۶۴ قرار میدهیم. نرخ یادگیری 5e-5 (با توجه به اینکه داریم تنظیم مجدد میکنیم، نرخ یادگیری را کوچک قرار میدهیم تا مدل دچار تغییرات شدیدی نشود) و Weight Decay را ۰.۰۱ قرار میدهیم. همچنین از نرخ یادگیری متغیر مدل WarmUp استفاده میکنیم تا در ابتدا با نرخ یادگیری کوچکی شروع کرده که مدل اصلی را از دست ندهیم، سپس رفته رفته نرخ یادگیری را زیاد کند تا مدل روی دیتاست مورد نظر بهینه شده و در انتها دوباره نرخ یادگیری را کاهش دهد تا مدل پایدار بشود.

از بهینه سازی AdamW و تابع هزینه CrossEntropy استفاده میشود.

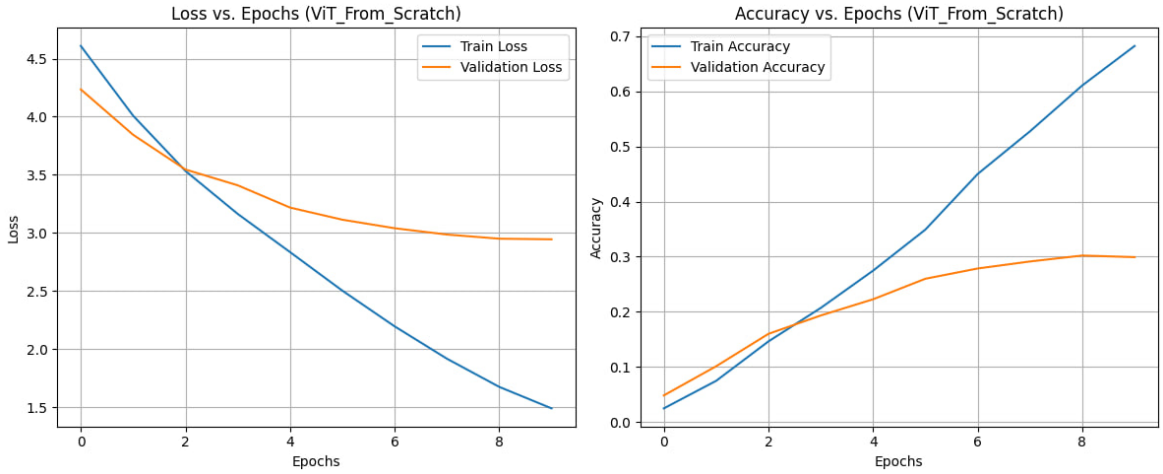


شکل ۱۲: نمودار دقت و خطای آموزش و اعتبارسنجی در تنظیم مجدد مدل

همانطور که میبینیم عملکرد مدل به صورت مناسبی با طی کردن ایپاک ها بهبود یافته است. دقت نهایی مدل روی داده های تست به ۸۳.۱۸ درصد می‌رسد.

۳و۴ : آموزش مدل بدون وزن های پیش‌آموزش دیده

این بار مدل را با وزن های اولیه رندوم بارگذاری کرده و آن را از پایه روی دیتاست مان برای ۱۰ ایپاک آموزش می‌دهیم. (با همان تنظیمات هایپرپارامتر قبل)



شکل ۱۳: نمودار دقت و خطای آموزش و اعتبارسنجی درآموزش مدل از پایه

در این حالت مشاهده میکنیم که دقت مدل روی داده های آموزش تا حد خوبی بالا رفته است ولی دقت روی داده های اعتبارسنجی پس از چند ایپاک به یک Plateu رسیده و دیگر بهبود نیافته است که نشانه ای از بیش‌برازش مدل می‌باشد. همانطور که قبلا هم اشاره شد با توجه به حجم بالای مدل و معماری ترنسفورمر که ذاتا به داده های بسیار زیادی برای یادگیری مناسب احتیاج دارد، تعداد داده های آموزش این دیتاست برای آموزش این مدل از پایه کافی نیست.

دقت مدل نهایی روی داده های تست ۲۵.۸۷ درصد می‌رسد.

یک بار نیز مدل را با نرخ یادگیری بزرگتر (5e-4) آموزش دادیم که بهبودی در یادگیری آن مشاهده نشد و دقت نهایی نیز دچار افت شد.

۵: تحلیل عملکرد دو مدل

مدل Vision Transformer استفاده شده روی ImageNet آموزش دیده شده است. این دیتاست شامل ۱۴ میلیون تصویر از ۱۰۰۰ کلاس مختلف است! این حجم بالای داده باعث میشود مدل به خوبی قادر به تشخیص ویژگی های مختلف تصاویر بشود و درک خوبی از تصاویر جدید نیز دارد و این دانش نهفته در مدل پیش آموزش دیده وجود دارد که با بهره گیری از روش Transfer Learning و تنها تنظیم دقیق مدل روی دیتاست مورد نظر توانستیم از این درک مدل بهره ببریم و به دقت بالایی روی یک دیتاست جدید (با هزینه محاسباتی پایین) برسیم.

ولی دیتاست Flowers-102 همانطور که گفته شد تنها حدود ۱۰۰۰ تصویر آموزشی دارد که اصلا برای یادگیری همچین مدل بزرگی کافی نیست و با آموزش مدل نمیتواند ویژگی های عمیق تصاویر را یادبگیرد و تنها برخی از ویژگی های سطحی را حفظ میکند. برای همین دچار بیش‌برازش شده و عملکرد ضعیفی نیز از خود نشان می‌دهد.

۳-۱: حملات متخاصم و دفاع

در این بخش مدل های بخش های پیشین را تحت حملات متخاصم FGSM و PGD قرار میدهیم و عملکردشان را بررسی میکنیم. سپس مدل ها را مقاوم سازی کرده و دوباره عملکردشان را تحلیل میکنیم.

روش های تولید نمونه های تخاصمی برای تولید یک نویز کوچک (و معمولا نامحسوس برای انسان) استفاده می‌شوند که با اعمال آن به تصویر اصلی، تابع خطای شبکه عصبی بیشترین افزایش را دارد تا مدل را به اشتباه وادارد.

روش های FGSM و PGD هر دو Foundational Gradient-based attack هستند، به این معنا که از گرادیان های خود مدلی که قصد حمله به آن را داریم استفاده میکنند.

روش FGSM (Fast Gradient Sign Method):

یک روش تک مرحله ای و سریع برای بدست آوردن نویز می‌باشد. نحوه عملکرد آن به این صورت هست که ابتدا Loss را محاسبه میکند. سپس گرادیان Loss را نسبت به تک تک پیکسل های تصویر ورودی محاسبه میکند. این روش تنها از علامت گرادیان ها استفاده میکند و یک ماتریس می سازد که هر درایه آن مثبت یا منفی یک است (با توجه به علامت گرادیان) و جهت بهینه تغییر برای هر پیکسل را نشان میدهد. در نهایت با ضرب این ماتریس در مقدار کوچک اپسیلون (که حداکثر مقدار مجاز نویز است) و افزودن آن به تصویر، نمونه متخاصم را می‌سازیم.

$$x_{adv} = x + \varepsilon * sign(\nabla_x J(\theta, x, y))$$

این روش فرض میکند که با برداشتن یک گام نسبتا بزرگ در جهت گرادیان میتوان تابع هزینه را به مقدار زیادی افزایش داد.

روش PGD (Projected Gradient Descent):

یک روش Iterative می باشد. نحوه عملکرد آن مشابه FGSM هست ولی با این تفاوت که در چند مرحله اجرا شده و هر مرحله گام کوچکی برمی‌دارد. بعد از برداشتن هر قدم هم مطمئن میشود که از محدوده مجاز ε خارج نشده باشد.

برای شروع معمولا به تصویر اولیه یک نویز رندوم کوچک اضافه می کنند تا از گیر کردن مدل داخل یک Local Optimum جلوگیری کند. سپس به تعداد Steps مراحل زیر را تکرار می‌کنیم:

ابتدا مانند روش FGSM گرادیان را برای هر پیکسل محاسبه کرده و علامت آن را تعیین میکنیم. سپس آن را در یک ضریب کوچک آلفا ضرب کرده و نویز حاصل را به تصویر اضافه می‌کنیم.

سپس بررسی میکنیم که نویز اضافه شده به تصویر از حد ε خارج نشده باشد که در این صورت دوباره نویز را به آن محدوده Project می‌کنیم.

این روش از نظر محاسباتی سنگین تر از روش قبل هست ولی به دلیل اینکه تعداد زیادی قدم کوچک در جهت افزایش Loss بر می‌دارد، نمونه های تخاصمی قوی تر را پیدا می‌کند و در عمل برای تولید تصاویر Adversarial از همین روش استفاده می‌کنند.

۱: پیاده‌سازی حمله FGSM و PGD

برای پیاده سازی این حملات از کتابخانه torchattacks و توابع آماده این حملات استفاده می‌کنیم.

```
fgsm_attack = torchattacks.FGSM(model_to_attack, eps=EPS_FGSM)
pgd_attack = torchattacks.PGD(model_to_attack, eps=EPS_PGD,
alpha=ALPHA_PGD, steps=STEPS_PGD)
```

پارامتر های حمله را هم مطابق صورت مسئله تعیین میکنیم.

تابع زیر را نیز برای بررسی هر مدل روی مجموعه تست متناظرش با اعمال حمله مینویسیم.

```
def evaluate_adversarial(model, dataloader, attack, model_type):
    model.eval()
    correct = 0
    total = 0
    pbar_desc = f"Evaluating {model_type.upper()} with
{attack.__class__.__name__}"
    for batch in tqdm(dataloader, desc=pbar_desc):
        if model_type == 'vit':
            images, labels = batch['pixel_values'], batch['labels']
        else:
            images, labels = batch
        images, labels = images.to(device), labels.to(device)
        adv_images = attack(images, labels)
        with torch.no_grad():
            outputs = model(adv_images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    return accuracy
```

۲: اعمال حملات متخاصم روی ۴ مدل بخش های قبل

در این مرحله هر دو حمله متخاصم را روی ۲ مدل ResNet بخش اول (مدل های آموزش داده شده روی تصاویر اصلی و تصاویر نویزی) و ۲ مدل ViT (مدل تنظیم مجدد شده و مدل آموزش داده شده از ابتدا) اعمال میکنیم. برای اینکار تصاویر تست هر یک از مدل ها را با اعمال حمله تغییر میدهیم و تصویر جدید را به مدل اعمال میکنیم و عملکرد مدل را ارزیابی میکنیم. جدول زیر نتایج را نشان میدهد.

۱- جدول ۱: عملکرد مدل ها برروی حملات **PGD** و **FGSM**

Model	Base Accuracy (%)	FGSM Accuracy (%)	PGD Accuracy (%)
ResNet18 (Original)	67.14	11.67	1.77
ResNet18 (Noisy)	37.04	10.03	7.94
ViT (Fine-tuned)	83.18	7.79	0.00
ViT (From scratch)	25.87	0.11	0.00

همانطور که میبینیم عملکرد همه مدل ها دچار افت بسیار فاحشی شده است ولی چند نکته نیز وجود دارد. اول اینکه همانطور که انتظار داشتیم حمله PGD به دلیل چندمرحله بودن و تولید نویز مخرب‌تر برای تصویر، اثر بسیار مخرب‌تری نیز روی عملکرد مدل میگذارد و در هر ۴ مدل این امر مشهود است.

نکته دوم عملکرد مدلی است که با داده های نویزی آموزش داده‌شده است. اگر چه این مدل هم دچار افت دقت قابل ملاحضه شده ولی نسبت افت دقت آن به دقت اولیه اش کمترین مقدار در بین ۴ مدل است. به عبارتی این مدل عملکرد بهتری روی حملات از خود نشان داده است (در حملات قوی PGD با اختلاف بهترین عملکرد را دارد). این امر به دلیل این است که این مدل ذاتا روی داده های نویزی آموزش دیده است و حدی از مقاومت نسبت به نویز را در خود ایجاد کرده است که در اینجا و با اعمال حمله به آن خود را به خوبی نشان میدهد.

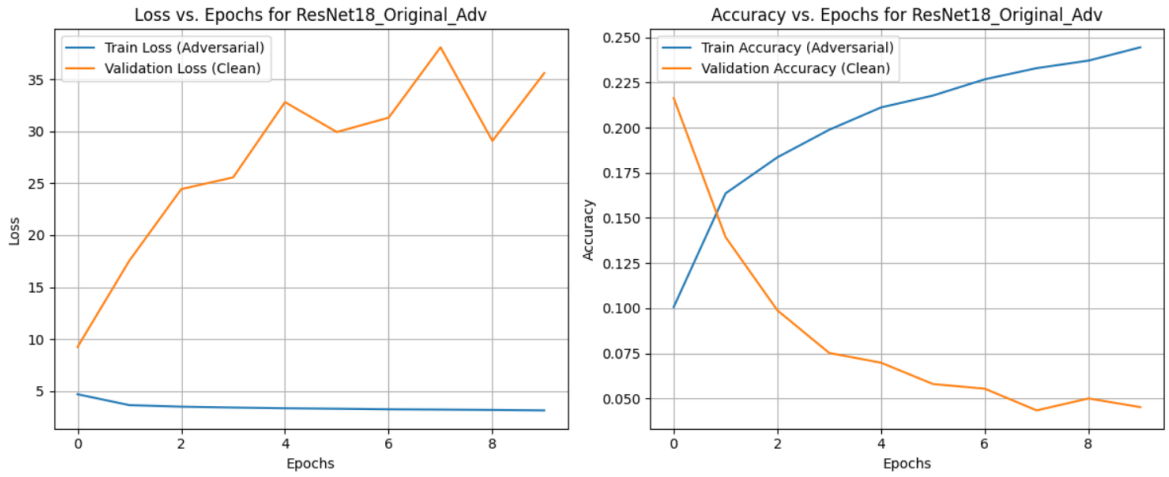
نکته دیگر عملکرد بسیار ضعیف مدل های ViT بعد از حمله است (دقت صفر برای حملات PGD !). دلیل این امر، یادگیری سطحی این مدل ها در مراحل آموزش است (به ویژه مدل آموزش داده شده از پایه) که باعث ناتوانی مدل ها در استخراج ویژگی های عمیق میشود. به همین دلیل با اعمال حمله و تغییر ویژگی ها مدل به کلی توانایی تشخیص خود را از دست میدهد.

۳: آموزش مجدد مدل ها (Adversarial training)

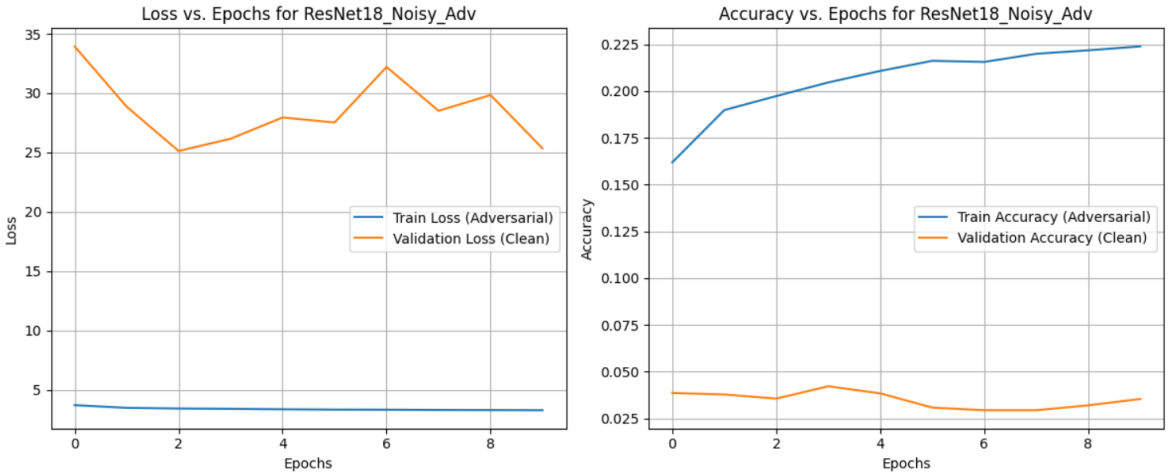
در این بخش هر ۴ مدل بخش های قبل را به مدت ۱۰ ایپاک برروی تصاویر متخاصم آموزش میدهیم تا مدل ها را نسبت به این تصاویر تهاجمی Robust کنیم.

نرخ یادگیری را در همه آموزش های زیر 1e-4 و Weigh Decay را ۰.۰۱ قرار میدهیم. پارامترهای حملات برای تولید نمونه های متخاصم را همان مقادیر بخش های قبل قرار میدهیم. (تنها Step های حمله PGD را ۴ قرار میدهیم تا فرآیند آموزش سریعتر پیش برود)

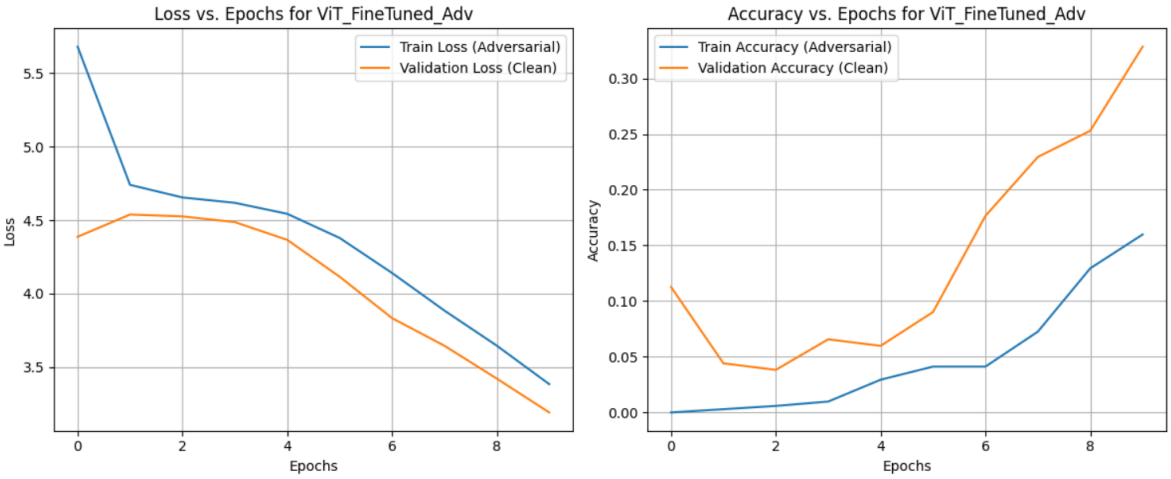
از اپتیمایز Adam و تابع هزینه CrossEntropy طبق معمول استفاده می‌کنیم.



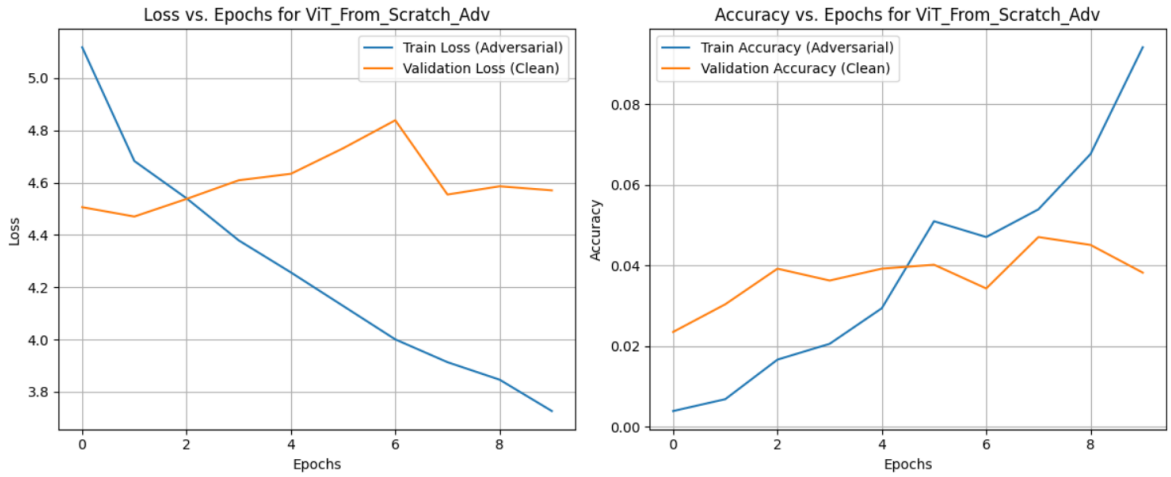
شکل ۱۴ : نمودار دقت و خطای آموزش و اعتبارسنجی مدل **ResNet Original** روی داده های تخاصمی



شکل ۱۵: نمودار دقت و خطای آموزش و اعتبارسنجی مدل **ResNet Noisy** روی داده های تخاصمی



شکل ۱۶: نمودار دقت و خطای آموزش و اعتبارسنجی مدل **ViT F.T** روی داده های تخاصمی



شکل ۱۷: نمودار دقت و خطای آموزش و اعتبارسنجی مدل **ViT Scratch** روی داده های تخاصمی

۴: مقایسه دقت مدل ها

در این بخش دقت مدل ها را به صورت جامع مقایسه میکنیم. مدل های آموزش داده شده بخش قبل را یکبار روی داده های تمیز و بار دیگر تحت حملات متخاصم ارزیابی میکنیم.

۲- جدول ۲: مقایسه جامع دقت مدل ها

Model	Base Accuracy (%)	FGSM Accuracy (%)	PGD Accuracy (%)	Clean Acc. (after adversarial training)	FGSM Acc. (after adversarial training)	PGD Acc. (after adversarial training)
ResNet18 (Original)	67.14	**11.67**	1.77	4.81	**44.57**	**24.23**
ResNet18 (Noisy)	37.04	10.03	**7.94**	3.24	28.76	16.96
ViT (Fine-tuned)	**83.18**	7.79	0.00	**25.43**	17.51	8.11
ViT (From scratch)	25.87	0.11	0.00	4.19	8.66	5.29

در بخش های قبل عملکرد مدل‌ها بدون حمله و تحت حملات متخاصم مقایسه و تحلیل شده بود. در اینجا بعد از آموزش مجدد مدل روی نمونه های متخاصم آموزشی، دقت آن را روی نمونه های تست سالم و نمونه های متخاصم شده با حملات FGSM و PGD مجدد محاسبه نموده‌ایم.

همانطور که در جدول ۲ میبینیم، بعد از آموزش متخاصم، دقت مدل ها روی داده های تست تمیز به شدت افت کرده است! این امر میتواند به خاطر آموزش ضعیف مدل ها در مرحله اول رخ دهد. درصورتی که مدل روی داده های زیاد و به صورت عمیق الگوها را یاد نگیرد، هنگام یادگیری متخاصم و با اعمال نمونه های متخاصم که تضاد زیادی با نمونه های سالم دارند (همانطور که در بخش های قبل توضیح داده شد این حملات به گونه طراحی میشوند که تابع هزینه مدل را بیشینه کنند)، دچار فراموشی الگوهای قبلی شده و الگوهای جدید متخاصم را سعی میکند فرا بگیرد. به همین دلیل عملکرد آن ها در طبقه بندی تصاویر سالم به شدت افت میکند. نکته قابل توجه این است که بهترین عملکرد را بعد از آموزش متخاصم با اختلاف زیادی مدل ViT تنظیم مجدد شده از خود نشان میدهد. این امر را میتوان با همان توضیحات بالا توجیه نمود، زیرا این مدل برروی حجم عظیمی از داده ها آموزش دیده است و به صورت عمیقی قادر به تشخیص ویژگی های تصویر است و با آموزش آن روی تعداد نسبتا پایینی از تصاویر متخاصم، مدل همچنان بخش زیادی از توانایی خود را حفظ میکند.

حال عملکرد مدل ها را پس از دفاع تحت حملات متخاصم بررسی میکنیم. همانطور که انتظار داشتیم تمامی مدل ها پس از آموزش، عملکرد بهتری تحت حملات از خود نشان دادند. در مدل های ResNet،

مدل اول (آموزش داده شده روی داده های بدون نویز) دقت خوب ۴۴ درصد را تحت حمله FGSM و دقت ۲۴ درصد را تحت حمله PGD از خود نشان میدهد.

در مدل های ViT، مدل تنظیم مجدد شده عملکرد بهتری نسبت به مدل آموزش داده شده از پایه دارد. ولی هر دو مدل تحت حملات عملکرد به مراتب بهتری نسبت به حالت قبل از دفاع از خود نشان میدهند.

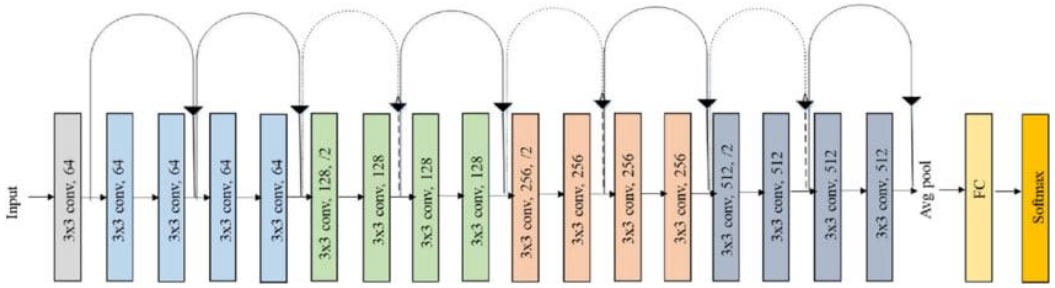
۴-۱: سوالات تئوری

پرسش ۱:

نکته اولی که باید به آن توجه داشت این است که نویز گوسی اگر چه اختلال زیادی در تصاویر ایجاد میکند، ولی برخلاف روش های متخاصم، اطلاعات اصلی تصویر را از بین نمی‌برد و مدل همچنان قادر به درک و استخراج الگوها از تصاویر می‌باشد.

همچنین به این نکته باید توجه کرد که واحد اصلی این شبکه ها لایه های کانولوشن هستند. این لایه ها با استخراج ویژگی های تصاویر مثل بافت، لبه ها و اشکال هندسی در سطوح مختلف، در فرآیند یادگیری از عوامل نایقینی و غیر مهم چشم پوشی میکنند و با فیلترهایی که دارند روی نواحی کوچکی از تصویر متمرکز میشوند. پس نویز گوسی که به صورت رندوم به برخی پیکسل ها اضافه میشود را این لایه ها میتوانند نادیده بگیرند. (به صورت بنیادی ویژگی شبکه های CNN مقاومتشان در برابر نویز و نایقینی‌ها می‌باشد)

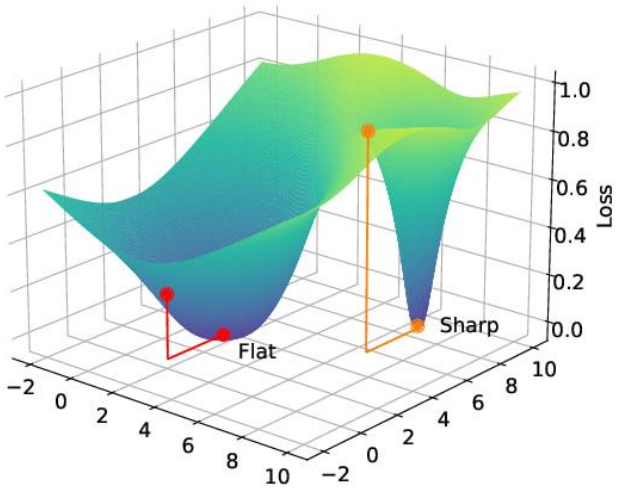
علاوه بر آن برخی از مکانیسم های مدل ResNet مانند Batch Normalization، Dropout و Residual Connection (که اطلاعات لایه های اولیه را به لایه های عمیق تر مستقیم انتقال میدهد) باعث مقاومت آن در برابر نایقینی ها و به طور خاص داده های نویزی میشوند.



شکل ۱۸: ساختار **ResNet18** و عملکرد مکانیسم **Residual Connection**

پرسش ۲:

در فضای خطای یک مدل، هر نقطه نشان دهنده یک مجموعه از پارامترها هست که یک مقدار خطا (Loss) را نتیجه میدهد. بهینه ساز در فرآیند یادگیری سعی میکند نقطه ای را پیدا کند که کمینه خطا را داشته باشد. در اینجا دو مفهموم مینیمم تخت و تیز به وجود می‌آید.



شکل ۱۹: مقایسه مینیمم تخت و تیز

همانطور که در شکل ۱۹ میبینیم، مینیمم تخت مانند یک دره وسیع است که اگر نقطه ای در این دره را در نظر بگیریم، با کمی جابجا شدن (تغییر وزن های مدل)، خطای مدل تغییر آنچنانی نمی‌کند و همچنان عملکردی مشابه قبل دارد. این ویژگی باعث تعمیم پذیری بالای مدل می‌شود، زیرا مدل نه تنها روی داده های آموزشی به نقطه قابل قبولی رسیده، بلکه روی داده های جدید که در فضای خطا میتوانند نقطه بهینه را کمی جابجا کنند نیز همچنان عملکرد خوبی دارد.

در مقابل مینیمم تیز وجود دارد که مانند یک گودال عمیق است که در نقطه پایینی آن خطای مدل کم است ولی در صورت جابجایی کوچک، مدل به شدت از نقطه بهینه فاصله میگیرد و عملکرد خوبی از خود نشان نمیدهد. این باعث تعمیم پذیری پایین مدل روی داده های جدید (تست) میشود، گویی که مدل تنها داده های آموزش را حفظ کرده و توانسته در فضای چند بعدی ویژگی ها یک مینیمم تیز پیدا کند.

مدل ViT از پیش آموزش دیده روی حجم عظیمی از داده ها، در فضای خطا مدل را در نقطه خوب و تعمیم پذیری قرار میدهد (مینیمم تخت) و مدل ویژگی های عمیق را فرا گرفته است. به همین دلیل با اعمال تنظیم مجدد به مدل و تغییر کوچک پارامترها برای سازگاری با مسئله جدید، مدل از این مینیمم تخت خارج نمیشود و همچنان عملکرد قابل قبولی از خود نشان میدهد. ولی مدلی که Pretrain نشده

است و از ابتدا روی دیتاست بسیار کوچک آموزش داده میشود، تنها میتواند این داده های آموزشی را تا حدی حفظ کند و در عمل یک نقطه مینیمم تیز و ناپایدار پیدا کند. برای همین ویژگی های قابل تعمیمی یاد نگرفته و با اعمال داده های جدید به مدل، عملکرد آن به شدت کاهش می‌یابد.

پرسش ۳:

با توجه به اینکه هدف ما در هر دو روش بهبود تعمیم‌پذیری و Robustness مدل هست، میتوان Adversarial training را هم نوعی از Data Augmentation در نظر گرفت. هر دو روش تصاویر جدیدی تولید کرده و مدل را روی آن ها آموزش میدهند تا مدل بجای توجه به الگوهای سطحی و بی اهمیت و غیرتاثیرگذار در نتیجه نهایی، روی الگوهای عمیق و تاثیرگذار تمرکز کند.

البته تفاوت هایی نیز این دو روش با هم دارند. مهم ترین تفاوتشان این است که در روش های مرسوم Data augmentation، تغییراتی محتمل در دنیای واقعی و در شرایط نرمال روی تصاویر ایجاد میشود (مانند تغییر نور، زاویه و پرسپکتیو تصویر). این تغییرات در شرایط واقعی قطعا روی تصاویر به وجود می‌آیند و مدل تعمیم پذیری بهتری روی نمونه های واقعی پیدا میکند. ولی در نمونه های متخاصم، تغییرات غیرواقعی روی داده ها ایجاد میکنیم که مدل را عمدا به اشتباه وادار کنیم. در حقیقت این روش به دنبال ایجاد مقاومت برای مدل تحت شرایط خاص رخ دادن حمله های خارجی استفاده میشود نه برای تعمیم پذیری مدل روی داده های واقعی خارجی.

همچنین نحوه عملکرد Augmentation های مرسوم مستقل از مدل است و تنها روی داده ها اعمال میشود ولی برای تولید نمونه های متخاصم دسترسی به مدل ضروری است و داده های متخاصم با توجه به مدل (و برای فریب دادن آن به بیشترین حد ممکن) تولید می‌شوند.

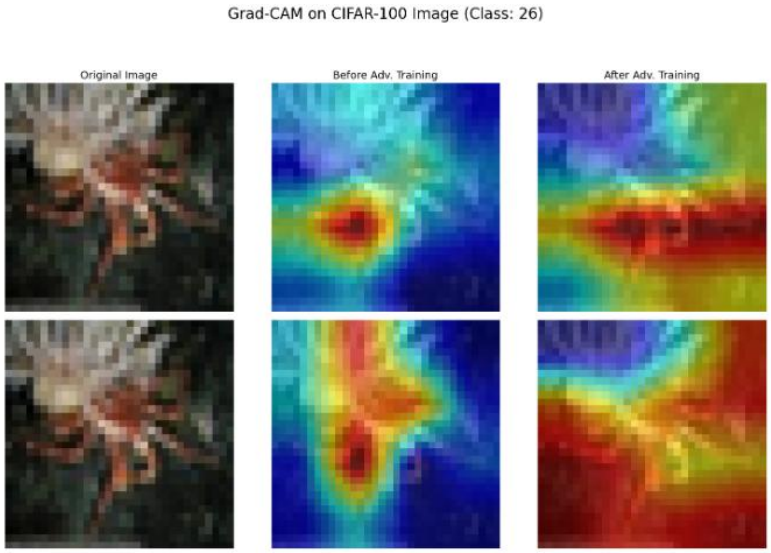
پرسش ۴:

همانطور که دیدیم عملکرد مدل های ResNet در برابر حملات متخاصم بهتر از مدل ViT بود. برای تحلیل این عملکرد از مفهموم Ensemble استفاده میکنیم. در تعریف کلاسیک منظور از Ensemble ترکیب چندین مدل (گاها با ساختارهای متفاوت) با یکدیگر برای دستیابی به عملکرد بهتر است. این مدل ها به روش های مختلفی میتوانند ترکیب شوند و برای مثال میانگینی از خروجی های آن ها برای تصمیم گیری استفاده شود. در معماری ResNet همان طور که اشاره شد از مکانیسم Residual/Skip connection استفاده شده است؛ به این صورت که برخی اتصالات وجود دارند که با ایجاد مسیرهای فرعی اطلاعات را به صورت مستقیم به لایه های جلوتر میفرستند. این کار باعث ایجاد مجموعه ای از مسیرهای موازی با طول های متفاوت در شبکه میشود که اطلاعات از آن ها عبور میکنند و به نوعی Ensemble در این مدل نهفته است.

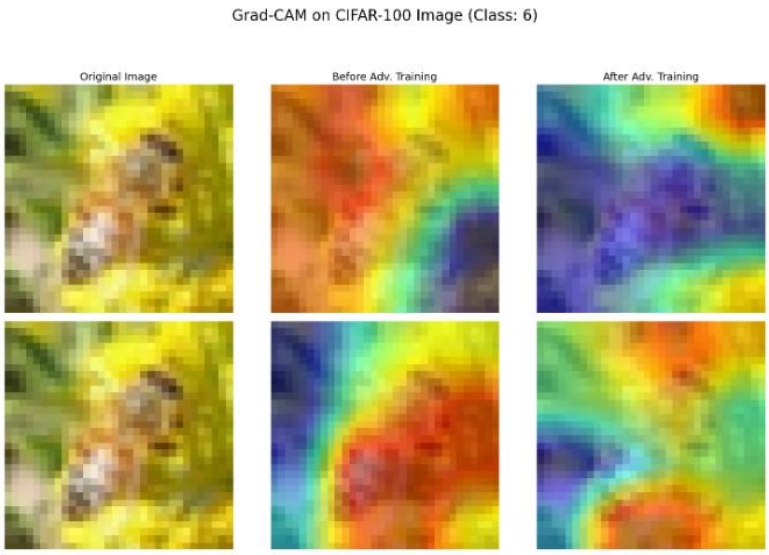
برای همین حتی در صورت مختل شدن اطلاعات تعداد از مسیرها تحت حمله تخاصمی، اطلاعات همچنان میتوانند از مسیرهای جایگزین به لایه های عمیق تر برسند. در اصل شبکه میتواند خروجی های تعدادی از بلوک ها که بیشتر تحت تاثیر نویز قرار گرفته‌اند را نادیده بگیرد (مانند Dropout عمل میکند) و از انتشار آن ها به لایه های بعدی جلوگیری کند. ولی در ساختار ViT یک Self-attention بین پچ های مختلف تصویر ایجاد میشود که با هم تعامل میکنند و تاثیر مخرب نویز در یک پچ کاملا روی تمام پچ ها اثر مخرب میگذارد و بر خلاف ResNet در اینجا دیگر مکانیزم Ensemble برای نادیده گرفتن تاثیر مخرب وجود ندارد و یک مسیر اطلاعاتی واحد در شبکه وجود دارد.

۵-۱: امتیازی

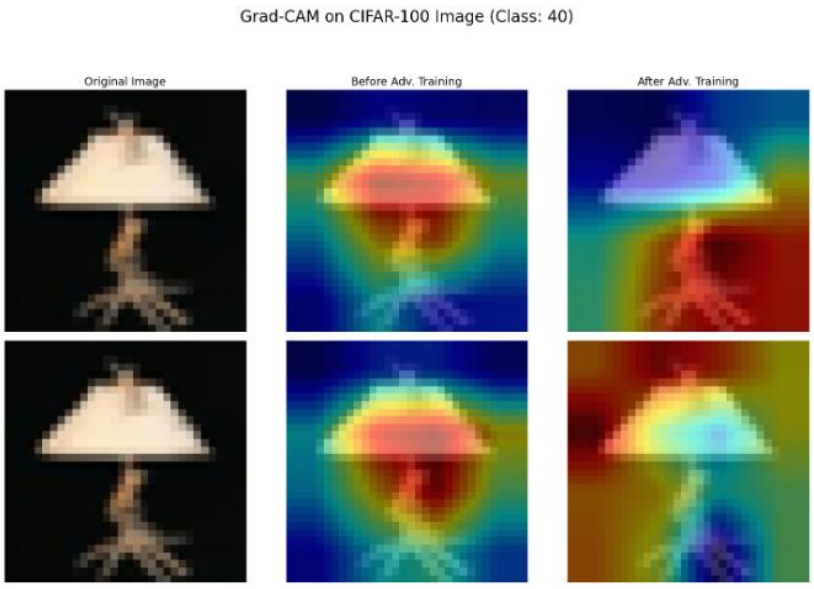
در این بخش برای ۴ مدل آموزش دیده اولیه و همچنین مدل های بعد از اعمال Adversarial training، نمونه هایی از ۴ کلاس مختلف را انتخاب میکنیم و نشان میدهیم مدل در کدام نواحی تصویر متمرکز شده است. این امر میتواند به مقایسه بهتر مدل ها قبل و بعد از Adversarial training کمک کند و دلیل افت دقت فاحش مدل ها روی داده های تمیز را بهتر درک کنیم.



شکل ۲۰: مقایسه نواحی تمرکز مدل های **ResNet** قبل و بعد از **Adversarial training** (نمونه ۱)



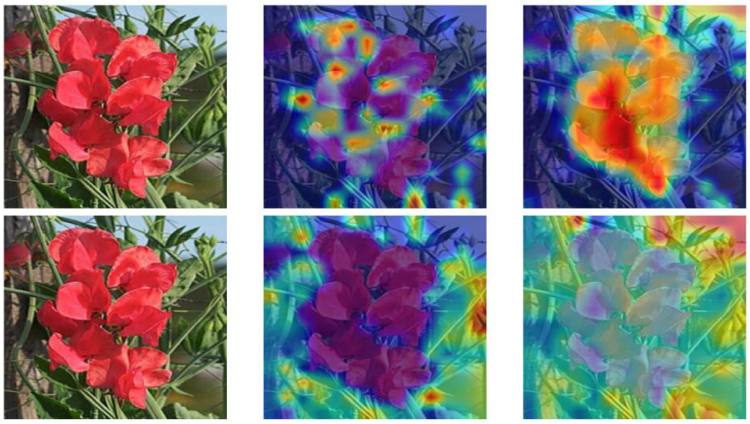
شکل ۲۱: مقایسه نواحی تمرکز مدل های **ResNet** قبل و بعد از **Adversarial training** (نمونه ۲)



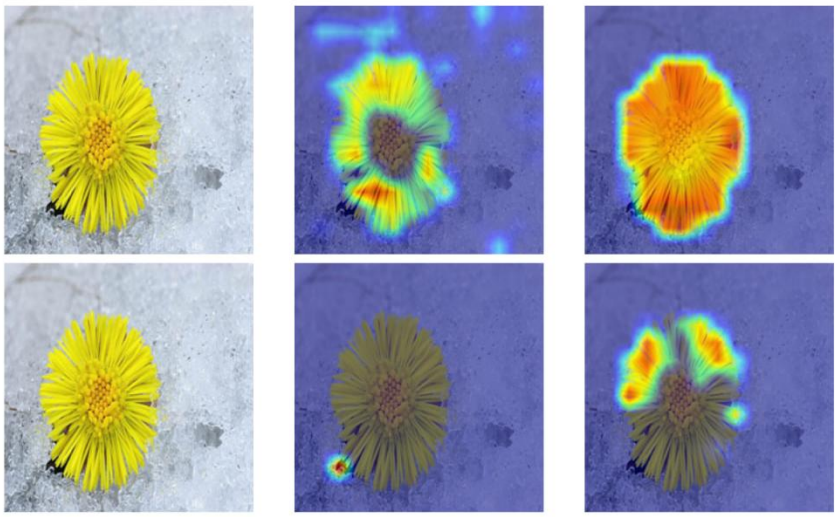
شکل ۲۲: مقایسه نواحی تمرکز مدل های **ResNet** قبل و بعد از **Adversarial training** (نمونه ۳)

در هر ۳ نمونه بالا ردیف اول مدل ResNet بدون نویز و ردیف دوم مدل با نویز را نشان میدهند. همانطور که به وضوح مشخص است بعد از آموزش تخاصمی، مدل به شدت در تشخیص نواحی مهم تصویرکه شی در آنجا قرار دارد دچار خطا میشود و بیشتر روی حاشیه های تصویر متمرکز می‌شود. دلیل افت عملکرد فاحش مدل ها روی داده های تمیز بعد از دفاع را اینگونه میتوان توجیه نمود، زیرا مدل دیگر نمیتواند روی ویژگی های واقعی تصویر تمرکز کند. همچنین عملکرد اولیه هر دو مدل و نواحی تمرکزشان شباهت زیادی به یکدیگر دارند.

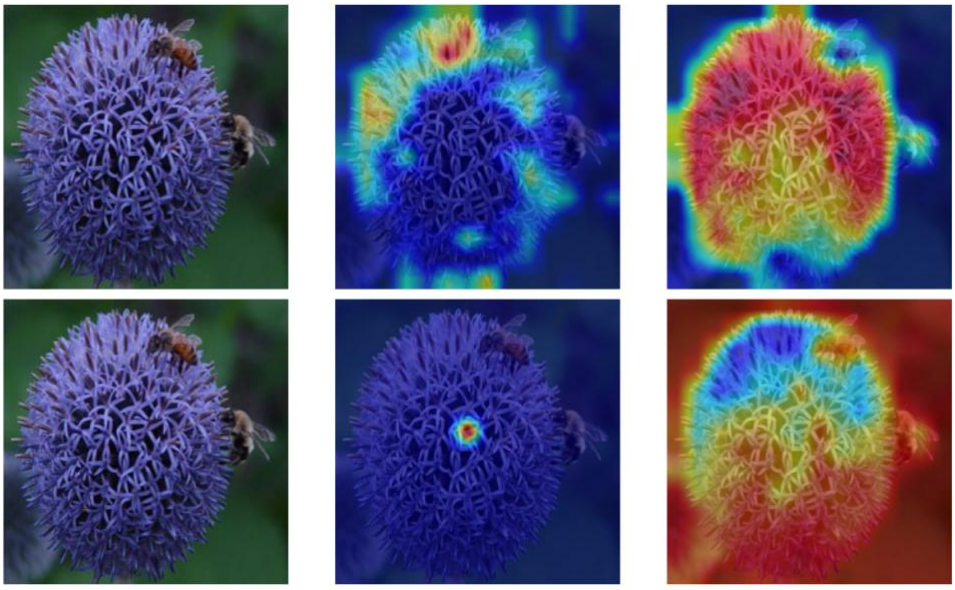
حالا عملکرد مدل های ViT را بررسی می‌کنیم.



شکل ۲۳: مقایسه نواحی تمرکز مدل های **ViT** قبل و بعد از **Adversarial training** (نمونه ۱)



شکل ۲۴: مقایسه نواحی تمرکز مدل های **ViT** قبل و بعد از **Adversarial training** (نمونه ۲)



شکل ۲۵: مقایسه نواحی تمرکز مدل های **ViT** قبل و بعد از **Adversarial training**
(نمونه ۳)

در تصاویر بالا ردیف اول نشان دهنده مدل تنظیم دقیق شده و ردیف دوم مدل آموزش داده شده از پایه را نشان میدهند. نکته اول تفاوت ناحیه تمرکز این دو مدل با همدیگر هست، در حالی که مدل Pretrain شده به وضوح روی ویژگی های عمیقتر و بنیادی تری در این نمونه ها تمرکز کرده است، مدل آموزش داده شده بیشتر روی نواحی حاشیه ای و ویژگی های کم اهمیت که قابلیت کلاس بندی تصویر را ندارند تمرکز کرده است. به همین دلیل هم مدل اول به نتایج بسیار بهتری دست یافته بود.

همچنین می‌بینیم که مدل Pretrain شده بعد از Adversarial training هم تمرکز خوبی را روی نواحی مهم تصویر دارد و قابلیت تشخیص خود را از دست نداده است (کمی غیر متمرکزتر شده است ولی همچنان گل ها را شناسایی میکند و احتمالا در تشخیص ویژگی های بخصوص کلاس ها نتواند به خوبی عمل کند). در نمودار خطای آموزش این مدل (شکل ۱۶) هم دیدیم که در ۱۰ ایپاک آموزش داده شده، عملکرد مدل همواره بهتر میشد، پس میتوان حدس زد که احتمالا با افزایش تعداد ایپاک های آموزش میتوانستیم حتی به دقتی بهتر از ۲۵ درصد بعد از دفاع نیز برسیم. در کل قابلیت تعمیم مدل های پیش آموزش دیده و مفهوم مینیمم تخت در اینجا به خوبی مشاهده میشود.

مدل آموزش داده شده هم مطابق انتظار نواحی تمرکز خوبی ندارد (بعد از آموزش متخاصم) که نتیجه ضعیفی که بدست آورده بودیم (دقت حدود ۴ درصد) را توجیه می‌کند.

پرسش ۲ – تولید توضیحات متنی برای تصاویر

۲-۱: آماده سازی داده

در این سوال از تمرین می خواهیم از شبکه های عصبی و مکانیزم های توجه و معماری های ترنسفورمری استفاده کنیم و از آنها برای تولید توضیحات متنی برای تصاویر استفاده کنیم.

در ابتدا مجموعه داده COCO Captions را روی حافظه لوکال کولب لود می‌کنیم. در ادامه باید پیش پردازش هایی را روی تصاویر و کپشن ها اعمال کنیم:

- **پیش پردازش های روی تصویر:**

۳- با توجه به اینکه بسیاری از شبکه های کانولوشنی شناخته شده با اندازه تصاویر ۲۲۴ در ۲۲۴ آموزش دیده اند ما نیز انداز تصاویر را به ۲۲۴ در ۲۲۴ تغییر میدهیم. برای resize کردن از الگوریتم INTER_CUBIC استفاده میکنیم. در ابتدا ممکن است بنظر برسد این الگوریتم تصویر را تیز تر میکند و کیفیت آن را خراب میکند ولی این الگوریتم نسبت به الگوریتم های دیگر اطلاعات بیشتری را حفظ میکند. همچنین مقادیر پیکسل ها را بین ۰ و ۱ استاندارد سازی میکنیم.

- **پیش پردازش های روی کپشن ها :**

۴- برای پیش پردازش علائم نگارشی مانند ، ؟ ! . و علائم خاص مانند # @ $ % را حذف میکنیم. زیرا هدف نهایی ما تولید توصیفی از تصویر است و این اطلاعات کمک چندانی به ما نخواهند کرد. ولی اعداد را نگه می داریم و صرفا آنها را استاندارد سازی میکنیم. از آن جایی که ممکن است در متن ما هم اعداد فارسی و هم اعداد انگلیسی وجود داشته باشند، ما اعداد را تمام به یک نوع تبدیل می کنیم زیرا مفهوم اعداد انگلیسی و فارسی برای ما یکسان است. همچنین برای جلوگیری از به هم چسبیدن کلمات در Normalizer قرار میدهیم : correct_spacing=False

نکته دیگر در پیش پردازش متن ها این است که ما کلمه هایی که کمتر از سه بار در کل متن ها تکرار شده اند را حذف میکنیم و با توکن <unk> این کار از بروز غلط املایی جلو گیری میکند و نمی گذارد کلمات خاص و عجیب در دیکشنری وارد شوند. ولی با توجه به خواسته های سوال تعداد این کلامات را می شماریم تا به خواسته های آماری سوال پاسخ دهیم.

از توکن های <sos> و <eos> نیز به ترتیب برای ابتدا و انتهای جمله استفاده میکنیم. از آنجا که میخواهیم این کپشن ها را ذخیره کنیم فعلا از توکن های <pad> استفاده نمی کنیم زیرا باعث زیاد شدن حجم کپشن ها می شود. ولی وقتی خواستیم داده ها رو لود کنیم از این توکن برای

یکسان سازی طول کپشن ها استفاده خواهیم کرد. در ادامه چند تصویر و کپشن های متناظر آن‌ها نشان داده شده.

یک هواپیمای جت آبی که بر فراز یک شهر پرواز می کند.

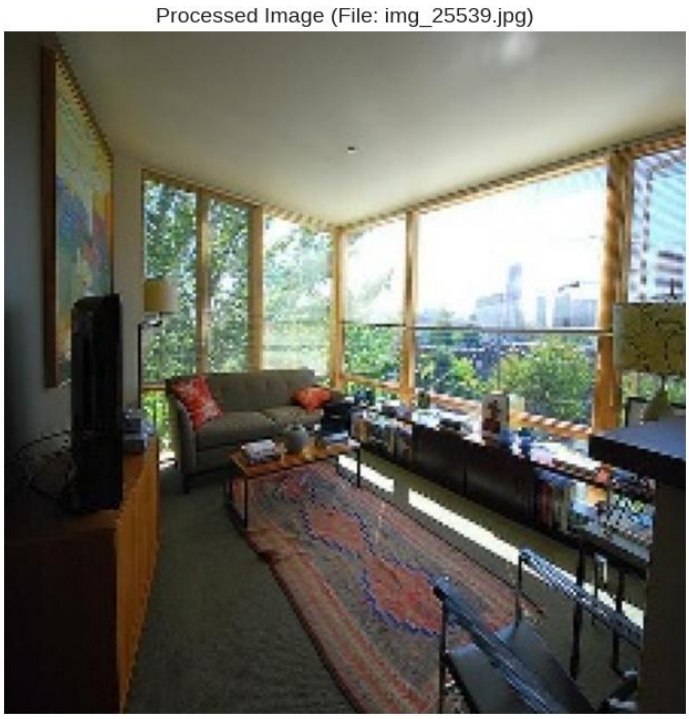
زنی که پشت یک گاو در مزرعه ایستاده است

زنی تنیس بازی می کند و آماده تاب خوردن است.

شکل ۲۶ تصاویر نمونه همراه با کپشن ها

در ادامه در شکل۲ یک نمونه داده همراه با کپشن عددی و دیکشنری مربوط به آن عدد ها آورده شده است.(علت افت کیفیت تصاویر نرمالایز کردن پیکسل ها و resize کردن آنها است.) همچنین در شکل۳ هیستوگرام طول کپشن ها و سپس کلمات پر تکرار همراه با تعداد کلمات یکتا آورده شده. مراحل تقسیم داده ها به بخش های train,test,validation و همچنین یکسان کردن طول کپشن ها با توکن <pad> را در بخش های بعدی انجام خواهیم داد.

یک نمونه تصویر با کپشن عددی:

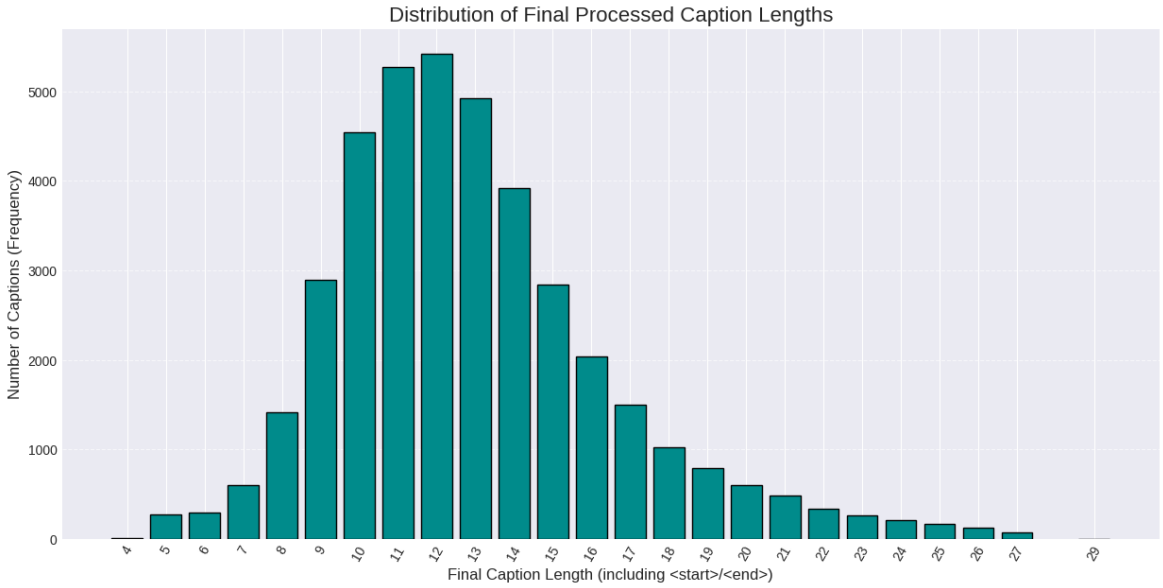


شکل ۲۷ نمونه تصویر با کپشن عددی

جدول ۳دیکشنری مربوط به ایندکس های این کپشن

Index	Word
1	<start>
12	به
160	نظر
53	می
161	رسد
132	روز
197	بسیار
1067	خوبی
20	در
189	بیرون
24	است
2	<end>

تحلیل های آماری:



شکل ۲۸ هیستوگرام طول کپشن ها

```
==================================================
=== Full Text Data Analysis (Before Filtering) ===
==================================================

Total number of unique words (before filtering): 9063

Top 10 most frequent words are:
  1. 'در': ۳۰۰۶۱ times
  2. 'یک': ۲۴۲۶۲ times
  3. 'با': ۱۳۲۱۶ times
  4. 'که': ۱۲۵۸۱ times
  5. 'و': ۱۲۳۵۱ times
  6. 'از': ۸۹۴۵ times
  7. 'روی': ۸۴۳۸ times
  8. 'حال': ۶۰۱۲ times
  9. 'مردی': ۴۹۶۶ times
  10. 'است': ۴۸۸۴ times
```

۲-۲: پیاده‌سازی معماری CNN + RNN با مکانیزم توجه

در این بخش می خواهیم مدل مورد نظر و بررسی شده در مقاله را پیاده سازی کنیم. قدم به قدم این کار را انجام میدهیم.

a. پیاده سازی encoder :

برای پیاده سازی encoder ما از مدلی که در دستور کار پیشنهاد شده یعنی همان مدل EfficientNet-B7 استفاده میکنیم. خود مقاله از مدل VGGNet استفاده کرده است که نسبت به EfficientNet-B7 کمی قدیمی تر و ضعیف تر است. نکته مهم این است که ما باید از آخرین لایه

کانولوشنی این شبکه ها استفاده کنیم. آخرین لایه کانولوشنی مدل VGGnet داری ابعاد 14x14x512 است. یعنی انگار تصویر خود را به یک گرید ۱۴ در ۱۴ تقسیم کرده ایم و برای هر کدام از این بخش ها ۵۱۲ ویژگی را استخراج کرده ایم. ولی در مدل EfficientNet-B7 این ابعاد 7x7x2560 است. یعنی یک گرید ۷ در ۷ که هر بخش ۲۵۶۰ ویژگی دارد. بخش دیکودر یک بخش کاملا مجزا از lstm و مکانیزم توجه است و به صورت استاتیک یک بار برای هر تصویر در ابتدای کار اجرا میشود و یک خروجی به ما میدهد و سپس ما از این خروجی در مراحل بعد استفاده میکنیم.

b. Lstm و مکانیزم توجه:

ما از خروجی encoder که متشکل از یک شبکه CNN بود ۴۹ بردار ۲۵۶۰ بعدی داریم که در واقع نقش Key را برای مکانیزم توجه ایفا میکنند. بردار hidden state یا همان خروجی قبلی لایه lstm (که نقش query را ایفا میکند) وارد مکانیزم توجه میشود. این بردار به هر یک از ۴۹ بردار موجود در Key یک امتیاز نسبت میدهد و با عبور از یک تابع softmax این امتیاز ها به مقادیر احتمال تبدیل می شوند به نحوی که حاصل جمع آنها ۱ باشد. امتیاز مربوط به هر بردار(که یک عدد اسکالر است) در تک تک درایه های این بردار ضرب میشود و و نهایتا این ۴۹ بردار که در امتیاز های متناظر با خودشان ضرب شده اند، باهم جمع می شوند و ورودی بعدی lstm را تشکیل میدهند. به این بردار که خروجی مکانیزم توجه و وردی lstm است، context vector می گوییم. context vector وارد lstm می شود و lstm با توجه به آن hidden state بعدی را تولید میکند که این hidden state به عنوان ورودی بعدی مکانیزم توجه نیز به کار می رود. نهایتا lstm در هر لحظه سه ورودی را دریافت میکند که state hidden جدید را تولید کند: hidden state قبلی، context vector فعلی، و embedding کلمه قبلی. در واقع این سه بردار باهم کانکت میشوند و وارد lstm میشوند. دقت کنیم که این عمل تا زمانی که مدل کل کپشن را تولید کند و به توکن <end> برسد ادامه پیدا میکند.

بررسی کنید چه مکانیزم های توجهی در مقاله "Show, Attend and Tell" پیاده‌سازی شده است.

در مقاله دو نوع مکانیزم توجه معرفی و پیاده‌سازی شده است:

۱- **Soft attention** : این دقیقاً همان روشی است که تا حالا درباره آن صحبت کردیم. در این روش، ما یک میانگین وزن‌دار از تمام بردارهای ورودی را در نظر می‌گیریم. این روش نرم است چون به همه بخش‌ها یک وزن (هرچند کوچک) اختصاص می‌دهد. مزیت اصلی این روش این

است که کل فرآیند مشتق‌پذیر است و می‌توان آن را به راحتی با روش Backpropagation آموزش داد.

۲- **Hard Attention** : در این روش، به جای ساختن یک میانگین، مدل فقط یکی از بردارهای ورودی را انتخاب می‌کند و بقیه را کاملاً نادیده می‌گیرد. این انتخاب بر اساس وزن‌های توجه (که به عنوان احتمال در نظر گرفته می‌شوند) به صورت تصادفی انجام می‌شود .از آنجایی که این فرآیند نمونه‌برداری مشتق‌پذیر نیست، آموزش آن پیچیده‌تر است و با روش Backpropagation قابل انجام نیست.

نکته مورد توجه محاسبه امتیاز مربوط به هر یک از بردار های Key است. در ساده ترین حالت بردار query تک به تک در این بردار ها ضرب داخلی می شود تا امتیاز مربوط به آنها را تولید کند. ولی در اکثر مواقع ابعاد بردار های موجود در key و بردار query یکسان نیست. برای حل این مشکل همانند چیزی که در صورت سوال گفته شده از روش Additive Attention استفاده می کنیم.

در روش Additive Attention یک لایه خطی داریم که تک تک بردار های موجود در Key را به یک بُعد میانی مثلا ۲۵۶ تبدیل میکند.(تک تک این بردار ها از یک لایه خطی یکسان عبور میکنند) حالا بجای ۴۹ بردار ۲۵۶۰ بعدی، ۴۹ بردار ۲۵۶ بعدی داریم. یک لایه خطی دیگر نیز بردار خروجی lstm که همان query است را به همان بعد میانی ۲۵۶ تایی تبدیل میکند. پس از این مرحله بردار query که از شبکه خطی عبور کرده را با تک تک بردار های Key جمع میکنیم. حالا ۴۹ بردار مثلا ۲۵۶ بعدی داریم که حاصل جمع بردار query با همان ۴۹ بردار قبلی است. پس از این مرحله هر ۴۹ بردار را از یک تابع تانژانت هایپربولیک عبور میدهیم و تک تک بردار ها را وارد یک شبکه عصبی میکنیم که ورودی آن ۲۵۶ تایی و خروجی آن یک عدد اسکالر است. انگار یک بردار قابل یادگیری است که در هر ۴۹ بردار ما ضرب داخلی می شود. پس از این مرحله ۴۹ عدد اسکالر تولید میشود که معرف ۴۹ امتیاز مربوط به ۴۹ بردار است و می توانیم پس از اعمال softmax روی آنها میانگین وزن دار بردار ها که همان خروجی مکانیزم توجه(context vector) است را حساب کنیم.

تا اینجا تقریبا تمام مفاهیم اصلی مقاله را بیان و بررسی کردیم. حالا می خواهیم چند مورد از نکات جزئی مقاله را بررسی کنیم و سپس شروع به پیاده سازی مدل کنیم:

۱- مقدار دهی اولیه lstm : در این مقاله مقادیر اولیه hidden state و cell state با مقادیر رندوم یا صفر مقدار دهی نمی شوند. در ابتدای کار میانگین تمام بردار های خروجی شبکه CNN گرفته

می شود و از دو شبکه MLP مجزا عبور می کند تا دو حالت صفر hidden state و cell state بدست آیند.

۲- لایه خروجی عمیق(Deep Output Layer) : اگر بخواهیم به طور معمول از یک شبکه lstm برای تولید متن استفاده کنیم، آخرین خروجی hidden state را به یک شبکه mlp میدهیم و خروجی این شبکه یک بردار n بعدی به ما میدهد که n تعداد کلمات موجو در دیکشنری ما است و هر یک از درایه های این بردار مشخص کننده احتمال تعیین کلمه مورد نظر است. ولی در این مقاله به این شکل رفتار نشده. در این مقاله hidden state فعلی، context vector و embedding آخرین کلمه تولیدی با هم جمع می شوند و به یک شبکه mlp داده می شوند تا خروجی پیشبینی شود.

۳- Embedding : ما برای embedding از fasttext استفاده نمی‌کنیم. علت این کار این است که تعدادی از کلمات در دیکشنری ما وجود دارند که در FastText وجود ندارد(برای مثال کلماتی در کپشن ها وجود داشتند که چسبیده بودند و جدا کردن دستی آنها ممکن نبود مانند خوشلباس ، موجسواری و...). برای اینکه بخواهیم از FastText استفاده کنیم باید این کلمات را خودمان لرن کنیم و بقیه کلمات را هم مجددا لرن کنیم. حالا در این مرحله اگر learning rate ما بالا باشد کلماتی که از قبل تیون شده بودند خراب می شوند و اگر learning rate ما پایین باشد همگرایی به شدت کند خواهد شد. (این پدیده صرفا یک نظر نیست و در نتایج کد دیده شد!) بنابراین ماتریس embedding را خودمان یادمی‌گیریم اگرچه در صورت وجود یک دیکشنری بهتر و یک کتابخانه قوی تر بهتر بود از آن استفاده کنیم.

۴- Gate : نکته دیگری که در مقاله مطرح شده است این است که به طور مستقیم از context vec در lstm استفاده نمی شود. در واقع context vec ابتدا در یک gate ضرب می شود و سپس وارد lstm می شود. این gate مقداری بین ۰و۱ دارد و از hidden state قبلی محاسبه می شود.

نهایتا با توجه به تمام این نکات مدل مورد نظر را پیاده سازی میکنیم. تعداد کل پارامتر های مدل و پارامتر ها قابل یادگیری آن در زیر آمده. (summary کامل مدل در notebook پس از آموزش نهایی مدل موجود است).

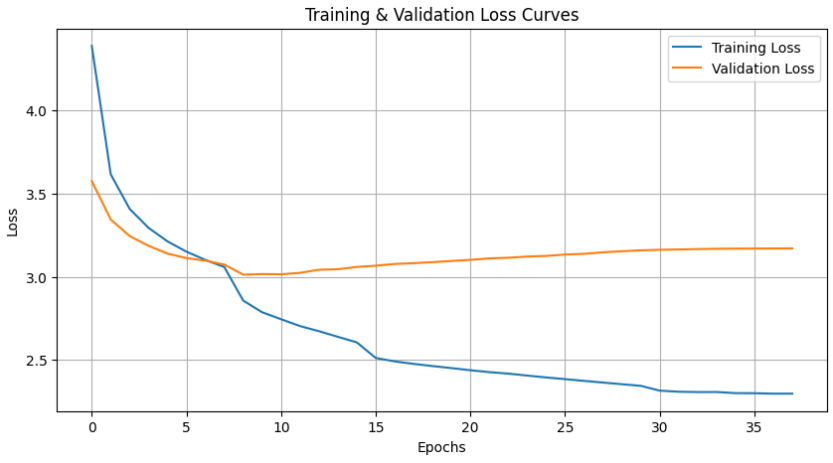
```
Total params: 78,164,585
Trainable params: 14,377,625
Non-trainable params: 63,786,960
Total mult-adds (Units.TERABYTES): 11.29
```

۲-۳: آموزش و ارزیابی

برای آموزش مدل از تابع خطای Cross Entropy و بهینه ساز adam استفاده میکنیم و مدل را روی تمام ۴۰۰۰۰ داده ترین میکنیم. از learning rate=0.001 استفاده میکنیم ولی اگر در ۳ ایپاک متوالی معیار BLEU-4 بهتر نشود مقدار learning rate را یک پنجم میکنیم. همچنین از early stop استفاده میکنیم که اگر به مدت ۱۲ ایپاک مقدار معیار BLEU-4 بهتر نشود یادگیری متوقف شود. از weight_decay = 0.00001 در بهینه سازی استفاده می کنیم که وزن های مدل بزرگ نشوند و مدل اور فیت نشود. همچنین از drop out ۶۰٪ برای لایه نهایی استفاده میکنیم. با این مشخصات مدل را به مدت ۶۴ ایپاک آموزش میدهیم. همچنین ازcheck point نیز استفاده شده تا اگر در هنگام آموزش مشکلی پیش آمد ادامه آموزش قابل انجام باشد. علت اینکه در نوت بوک آوزش از ایپاک ۱ شروعت نشده همین است.ابعاد میانی نیز در زیر آورده شده اند:

```
    'embed_dim': 256,
    'decoder_dim': 512,
    'attention_dim': 256,
    'encoder_dim': 2560,
```

در پایان هر ایپاک یک تصویر رندوم از مجموعه داده های اعتبار سنجی همراه با کپشن تولید شده توسط مدل و کپشن اصلی آورده شده که در notebook قابل مشاهده است. نمودار خطا روی داده های آموزش و اعتبار سنجی در زیر آمده است.



شکل ۲۹ نمودار خطا در طول آموزش روی داده های آموزش و اعتبارسنجی

نتیجه تست نهایی مدل بدین شکل است:

```
--- FINAL TEST RESULTS ---
Test Loss: 3.1399 | Test BLEU-1: 0.2311 | Test BLEU-4: 0.0385
```

مقاله برای ارزیابی از معیار های BLEU ۱ تا ۴ و همینطور معیار METEOR استفاده کرده. معیار های BLEU 1 و BLEU 4 ای که مقاله به آنها رسیده است تفاوت فاحشی با مقادیری که ما به آنها رسیدیم

دارد. از نظر من عمده این اختلاف به دلیل دیتاست و embedding است. در بخش تحلیل و بهبود مدل بیشتر در این باره صحبت میکنیم. نهایتا سه تصویر را با کپشن های اصلی و تولید شده توسط مدل نشان میدهیم.

original: دو گوسفند در حال چریدن روی یونجه و پوشاندن علف‌های یک مزرعه
Greedy: گوسفند در مزرعه ای در مزرعه است
Beam (k=3): دو گاو در یک مزرعه در حال خوردن علف هستند

شکل ۳۰ تصویر اول همراه با کپشن های تولید شده و کپشن اصلی

original: مردی در حال موج سواری روی تخته موج سواری در اقیانوس.
Greedy: مردی در حال موج سواری در موج سواری در اقیانوس
Beam (k=3): مردی در حال موج سواری در آب است

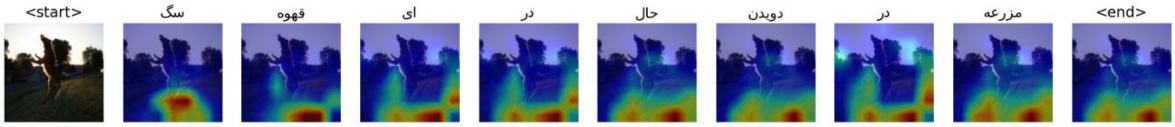
شکل ۳۱ تصویر دوم همراه با کپشن های تولید شده و کپشن اصلی

original: اواخر آروو در راه خانه توقف کردیم تا ببینیم چه چیزی در بالا داریم.
Greedy: برگ های سبز
Beam (k=3): برگ های سبز

شکل ۳۲ تصویر سوم همراه با کپشن های تولید شده و کپشن اصلی

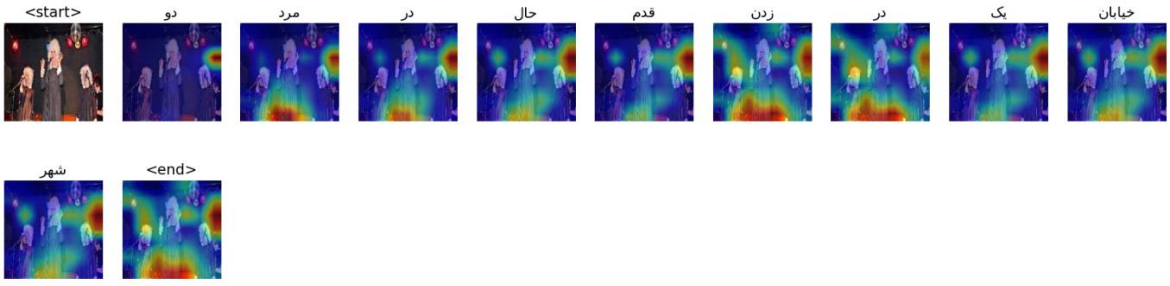
۲-۴: تحلیل و بهبود مدل

۱- تحلیل خطای مدل: مطابق دستور کار ۱۰۰ نمونه از داده ها را به صورت تصادفی انتخاب میکنیم و پنج تای بی کیفیت آنها را برمیداریم. سپس heatmap را روی تصاویر نشان میدهیم.



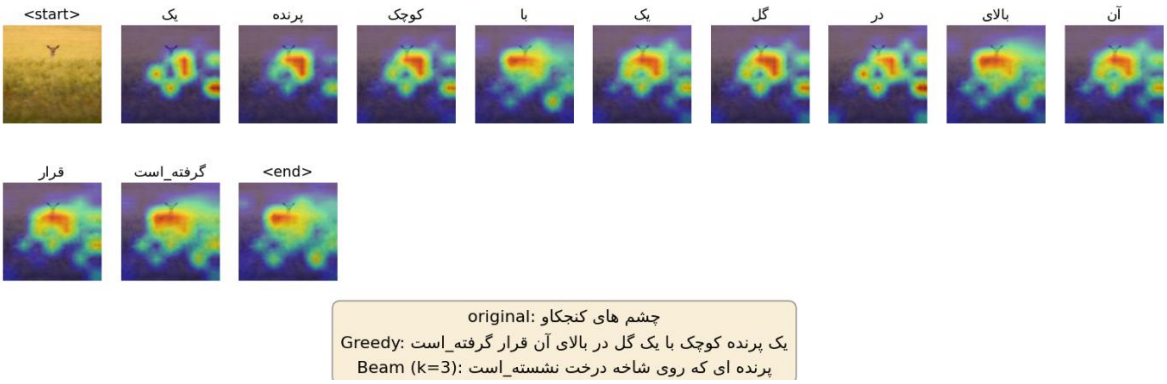
original: سگی برای گرفتن توپ تنیس به هوا می پرد.
Greedy: سگ قهوه ای در حال دویدن در مزرعه
Beam (k=3): یک سگ قهوه ای و سفید در حال بازی در چمن

شکل ۳۳ **heatmap** اول

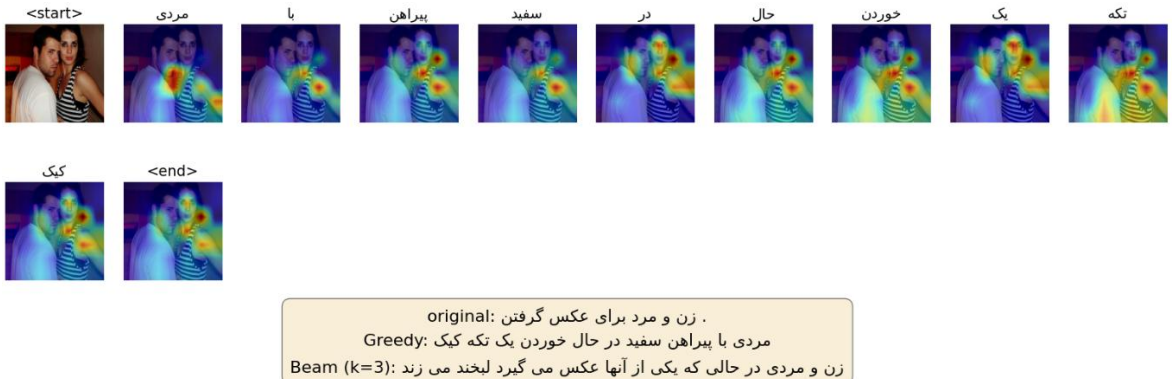


original: سه زن با لباس و کلاه گیس بلوند روی صحنه آواز می خوانند.
Greedy: دو مرد در حال قدم زدن در یک خیابان شهر
Beam (k=3): دو زن در حال قدم زدن در یک خیابان شهر

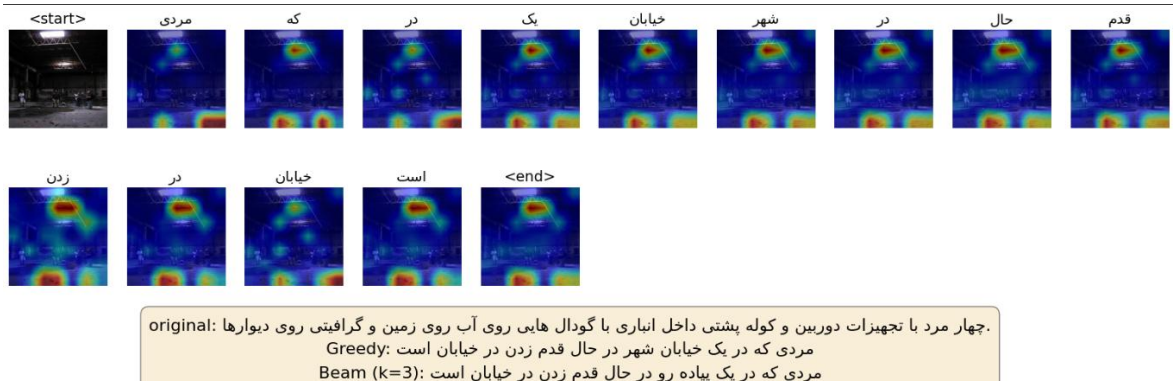
شکل ۳۴ **heatmap** دوم



شکل ۳۵ **heatmap** سوم



شکل ۳۶**heatmap** چهارم



شکل ۳۷ **heatmap** پنجم

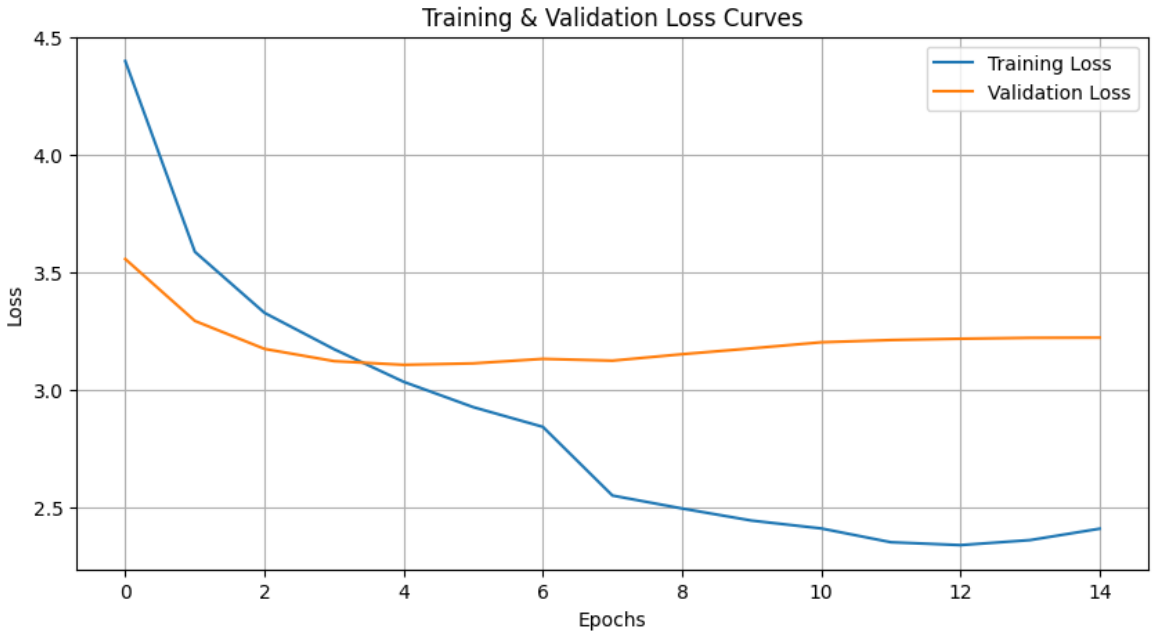
با توجه به این تصاویر می توان گفت در مواردی مدل به جاهای درست دقت کرده (برای مثال در تصویر اول) ولی در اکثر موارد مدل به جاهای درست دقت نکرده. به نظر من علت همه این موارد داده های بی کیفیت است! داده ها از چند جهت بی کیفیت هستند. یک مورد درباره کپشن های آنها است. بعضی کلمات بسیار انتزاعی هستن و مدل نمی توان آنها را بفهمد. برای مثال در همین نمونه ها تصویر سوم کپشن چشم های کنجکاو را دارد. ممکن است تصویر های زیادی شبیه همین تصویر در دیتاست ما وجود داشته باشند که کپشن های آنها اصلا هیچ ربطی به کلمات کنجکاو و چشم نداشته باشند و این امر موضوع را خراب میکند.

نکته دیگه ترجمه بد کپشن ها است. به عنوان مثال در یکی از تصاویر لیوان شیشه ای (glass) در فارسی به عینک ترجمه شده و در کپشن قرار گرفته بود. اگر تعدادی از تصاویر و کپشن های آنها را بازبینی کنید متوجه این موارد خواهید شد.

نکته دیگر resize کردن تصاویر است که باعث می شود کیفیت تصاویر کاهش یابد و ویژگی های معنادار خوب استخراج نشوند.

و نکته آخر و یکی از مهم ترین نکات ماتریس embedding است. این ماتریس به عنوان یکی از سخت ترین (و پر پارامتر ترین!) بخش های مدل وظیفه مهمی بر عهده دارد.map کردن کلمات به بردار ها در حالتی که کلمات متناظر بردار های نزدیکی به یکدیگر داشته باشند خود تسک بسیار دشوار و سختی است چه برسد به اینکه در کنار تسک سخت دیگری مثل image captioning انجام شود. متاسفانه با توجه به نکاتی که در بخش embedding مطرح شد ماتریس قوی ای برای این کار به زبان فارسی وجود ندارد که تمام لغات دیکشنری ما را پوشش دهد و همچنین کپشن های ما نیز تعداد زیادی کلمات خاص داشتند که در ماتریس embedding آماده FastText وجود نداشتند. از جمله کلمات بهم چسبیده. به همین دلیل ما مجبور بودیم که ماتریس embedding را خودمان یاد بگیریم و این کار را دشوار میکرد و کیفیت مدل را کاهش میداد.

۲- پیاده سازی Scheduled Sampling : در روش Teacher Forcing هنگامی که ما مدل را ترین میکنیم برای پیشبینی کلمه t ام از کلمه واقعی t-1 ام استفاده میکنیم. هنگامی که می خواهیم مدل را تست کنیم ما دیگر کلمه درست مرحله t-1 ام را نداریم و به همین دلیل مجبوریم از کلمه ای که خود مدل تولید کرده استفاده کنیم. در این حالت اگردر حالت تست مدل در یکی از کلمات ابتدایی جمله اشتباه کند این اشتباه به صورت آبشاری به بقیه کلمات منتقل می شود. برای حل این مسئله از یک رویکرد احتمالاتی استفاده می کنیم که در آن به احتمال p مدل از کلمه واقعی استفاده میکند و به احتمال 1-p از کلمه ای که خودش پیشبینی کرده استفاده میکند. اگر در ابتدا آموزش p=1 قرار دهیم و رفته رفته مقدار p را کاهش دهیم یعنی رفته رفته احتمال استفاده مدل از کلمات تولیدی خودش را افزایش میدهیم و در این صورت مدل می تواند خودش این مشکل را هندل کند. در کد قبلی این مکانیزم را اعمال میکنیم و p را از ۱ تا ۰.۵ به صورت خطی کاهش میدهیم. نتایج مطابق شکل صفحه بعد خواهد بود. انتظار میرود که وضعیت مدل بهتر شود ولی این امر مشهود نیست. علت آن این است که مقدار p به طور خطی از ۱ تا ۰.۵ در طول کل ۷۰ ایپاک آموزش ما کاهش می یابد ولی مدل ما early stop می شود. هنگامی که مدل early stop می شود مقدار p بسیار نزدیک به ۱ است و عملا تاثیر زیادی نمی گذارد.



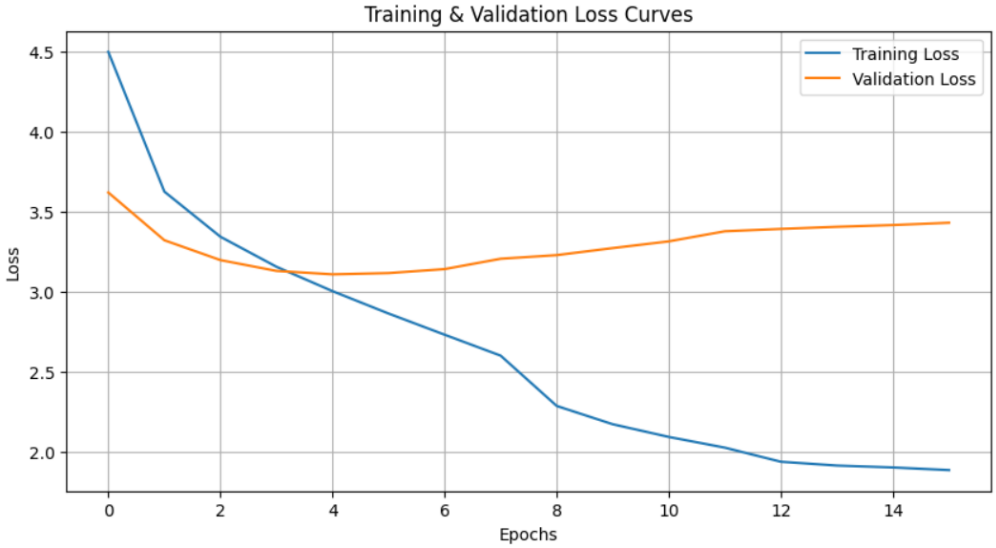
شکل ۳۸ نمودار خطا در حین یادگیری برای **Scheduled Sampling**

نتیجه تست نهایی:

```
--- FINAL TEST RESULTS ---
Test Loss: 3.1685 | Test BLEU-1: 0.2041 | Test BLEU-4: 0.0322
```

۳- آزمایش با مکانیزم های مختلف توجه:

مطابق دستور کار مکانیزم Scaled Dot-Product Attention را پیاده سازی میکنیم و مراحل آموزش و ارزیابی را تکرار می کنیم. نتایج نهایی به این شکل هستند.



شکل ۳۹ نمودار خطا در حین یادگیری برای مکانیزم **Scaled Dot Product Attention**

نتیجه تست نهایی:

```
---FINAL TEST RESULTS---
Test Loss: 3.1201 | Test BLEU-1: 0.2092 | Test BLEU-4: 0.0313
```

با توجه به این نتایج در حالت کلی نتایج این دو مدل و مدل اولیه تفاوت چندانی با هم ندارند و با توجه به معیار های BLEU می توان گفتن مدل اولیه به نسبت عملکرد بهتری داشته. تنها نکته مثبتی که این مکانیزم می تواند داشته باشد(که در اینجا آن را هم نداشته!) کم بودن پارامتر های آن و جلوگیری از اورفیت است.

در این مکانیزم توجه از ضرب داخلی برای سنجش میزان شباهت و از یک ضریب (همان $\frac{1}{\sqrt{d_k}}$) برای مقایس بندی استفاده می شود. نهایتا این میزان های شباهت پس از عبور از softmax به امتیاز تبدیل می شوند و می توان context vector را از رو ی آنها ساخت.