

به نام خدا

گزارش کارآموزی

نگارش:

سنانموحدین

دانشجوی دانشگاه امیرکبیر

باتشکر از:

جناب آقای دکتر خونساری

جناب آقای دکتر رستگار

جناب آقای فروغی

فهرست

۳	مقدمه
۴	دیتاست‌های استفاده شده
۴	MNIST دیتاست
۴	CIFAR-۱۰ دیتاست
۵	TENSORFLOW کتابخانه
۵	PYTORCH کتابخانه
۶	تسک اول
۶	CNN
۷	توضیح کد پیاده‌سازی شده با استفاده از کتابخانه TENSORFLOW
۹	تابع فعالسازی RELU و LEAKYRELU
۹	الگوریتم‌های بهینه‌سازی RMSPROP و ADAM
۱۰	نتایج تسک اول
۱۲	تجربیات تسک اول
۱۳	تسک دوم
۱۳	VGG
۱۳	CNN و VGG مقایسه‌ی
۱۶	نتایج تسک دوم
۱۶	تجربیات تسک دوم
۱۷	تسک سوم
۱۷	RESNET
۱۸	توضیح کد پیاده‌سازی شده با استفاده از کتابخانه PyTorch
۱۹	نتایج تسک سوم
۱۹	ResNet۱۸
۱۹	ResNet۱۶
۱۹	تجربیات تسک سوم
۲۰	تسک چهارم
۲۰	BACKDOOR ATTACK
۲۱	توضیح کد پیاده‌سازی شده
۲۲	نتایج تسک چهارم

مقدمه

امروزه در زمینه‌های مختلفی در جهان، هوش مصنوعی به طور گسترده‌ای استفاده می‌شود و کاربردهای آن روز به روز گستردگر هم می‌شود. یکی از مسائل مهم در مورد هوش مصنوعی، قابلیت انتکاپذیری و امنیت آن است. از یک جنبه، این که از چه مدلی استفاده می‌شود و آیا آن مدل قابل اعتماد هست یا خیر بسیار مهم است و از طرف دیگر با توجه به حجم زیاد داده‌ها و لزوم استفاده از داده‌های توزیع یافته در سیاری از بخش‌ها، مورد اعتماد بودن agent‌ها مطرح می‌شود.

بایدگیری فدرال (Federated Learning) یک روش بایدگیری ماشین است که به جای جمع‌آوری داده‌ها در یک سرور مرکزی، مدل‌ها را به دستگاه‌های محلی مانند گوشی‌های هوشمند ارسال می‌کند. این دستگاه‌ها مدل را با استفاده از داده‌های محلی خود آموزش می‌دهند و پس فقط وزن‌های به روزرسانی شده مدل را به سرور مرکزی ارسال می‌کنند. سرور مرکزی این وزن‌ها را جمع‌آوری کرده و با ترکیب آن‌ها، یک مدل به روزرسانی شده را به دستگاه‌ها بازمی‌گرداند. این فرآیند بدون انتقال داده‌های خام به سرور مرکزی، انجام می‌شود که به حفظ حریم خصوصی کاربران کمک می‌کند.

بایدگیری فدرال به ویژه در مواردی کاربرد دارد که داده‌ها به دلیل حساسیت یا حجم بالا قابل انتقال نیستند، مانند داده‌های پژوهشی یا داده‌های تولید شده توسط میلیون‌ها دستگاه کاربری. با این روش، علاوه بر کاهش تیاز به پهنای باند و حفظ حریم خصوصی، می‌توان از داده‌های توزیع شده در مقیاس بزرگ بهره‌برداری کرد و مدل‌های دقیق‌تر و بهینه‌تری ایجاد کرد. در طی این کارآموزی، هدف آشنایی بیشتر با مهمترین کتابخانه‌های مورد استفاده در تعریف مدل‌ها و همچنین آشنایی بیشتر با برخی مدل‌های مهم بود. همچنین در راستای امنیت در هوش مصنوعی، در تسک آخر به بررسی نوعی حمله به شکلهای عصبی پرداختم که در ادامه به طور مفصل تری توضیح داده می‌شود.

در تسک اول، از دیتاست MNIST و در دیگر تسک‌ها از دیتاست CIFAR-10 استفاده شده است؛ به همین دلیل ابتدا به توضیح مختصری درمورد هر یک از این دیتاست‌ها می‌پردازیم. همچنین با توجه به تاکید بر استفاده از دو کتابخانه‌ی مهم PyTorch و TensorFlow در انجام تسک‌ها، به این دو مورد هم در ادامه اشاره خواهیم کرد.

دیتاستهای استفاده شده

MNIST دیتاست

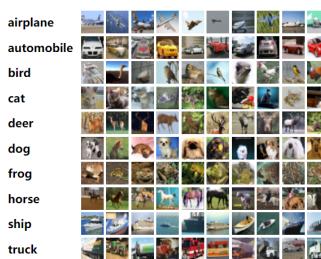
MNIST یک دیتاست مشهور و پرکاربرد در زمینه یادگیری ماشین و بینایی ماشین است که برای آموزش و ارزیابی الگوریتم‌های مختلف به ویژه شبکه‌های عصبی مورد استفاده قرار می‌گیرد. این دیتاست شامل ۷۰،۰۰۰ تصویر دستنویس از ارقام ۰ تا ۹ است که به صورت سیاه و سفید و با ابعاد ۲۸ در ۲۸ پیکسل ارائه شده‌اند. تصاویر به دو مجموعه آموزشی (۶۰،۰۰۰ تصویر) و تست (۱۰،۰۰۰ تصویر) تقسیم شده‌اند. دیتاست MNIST به دلیل سادگی و گستردنگی استفاده، به عنوان "Hello World" دنیای یادگیری ماشین شناخته می‌شود. از آنجایی که این مجموعه داده نسبتاً ساده است، بسیاری از مدل‌های پایه مانند شبکه‌های عصبی ساده (MLP) و شبکه‌های عصبی کانولوشنی (CNN) قادر به دستیابی به دقچهای بسیار بالا (حتی بالای ۹۹٪) بر روی این داده هستند.



نمودهای از دیتاست

CIFAR-10 دیتاست

این مجموعه هم یک دیتاست استاندارد و پرکاربرد در زمینه یادگیری ماشین و به ویژه بینایی ماشین است که برای آموزش و ارزیابی الگوریتم‌های مختلف، به خصوص شبکه‌های عصبی کانولوشنی (CNN)، استفاده می‌شود. این مجموعه داده شامل ۶۰،۰۰۰ تصویر رنگی در ۱۰ کلاس مختلف است. هر تصویر دارای ابعاد ۳۲ در ۳۲ پیکسل بوده و به یکی از کلاس‌های هوپیما، خودرو، پرنده، گربه، گوزن، سگ، قورباغه، اسب، کشتی و کامیون تعلق دارد. تصاویر به طور مساوی بین کلاس‌ها توزیع شده‌اند و مجموعه داده به دو بخش آموزشی (۵۰،۰۰۰ تصویر) و تست (۱۰،۰۰۰ تصویر) تقسیم شده است. CIFAR-10 به عنوان یک مجموعه داده پیچیده‌تر از MNIST. به دلیل تنوع کلاس‌ها و ویژگی‌های چالش‌برانگیز تصاویر، به محققان این امکان را می‌دهد تا الگوریتم‌ها و مدل‌های پیچیده‌تری را آزمایش کنند. این دیتاست به ویژه برای عملکرد شبکه‌های عصبی عمیق مناسب است و به عنوان یکی از معیارهای استاندارد برای مقایسه مدل‌ها در حوزه بینایی ماشین شناخته می‌شود. به دلیل اندازه کوچک تصاویر و تنوع بالا در دسته‌بندی، این دیتاست به یک چالش محبوب برای بهبود دقت مدل‌های یادگیری ماشین و یادگیری عمیق تبدیل شده است.



نمودهای از دیتاست

کتابخانه‌های مهم مورد استفاده

کتابخانه‌ی TensorFlow

پک کتابخانه منبع باز برای یادگیری ماشین و یادگیری عمیق است که توسط گوگل توسعه داده شده و به محققان و توسعه‌دهندگان این امکان را می‌دهد تا مدل‌های پیچیده‌ای از جمله شبکه‌های عصبی را به راحتی طراحی، آموزش و پیاده‌سازی کنند. این فریمورک از یک ساختار گراف محاسباتی استفاده می‌کند که در آن عملیات‌های مختلف به صورت گرافی از گره‌ها و لبه‌ها نمایش داده می‌شوند. این ویژگی باعث می‌شود که TensorFlow بتواند به طور کارآمدی بر روی سخت‌افزارهای مختلف، از جمله پردازنده‌ها (CPU) و واحدهای پردازش گرافیکی (GPU) اجرا شود و عملکرد بالای را ارائه دهد.

TensorFlow همچنین دارای ابزارها و کتابخانه‌های متنوعی است که فرآیند توسعه مدل‌های یادگیری عمیق را تسهیل می‌کند. به عنوان مثال، Keras API سطح بالا، به طور مستقیم بر روی TensorFlow ساخته شده است و به کاربران اجازه می‌دهد تا به سادگی و یا کد کمتر، مدل‌های پیچیده را طراحی و آزمایش کنند. با قابلیت‌های گسترده، از جمله پشتیبانی از یادگیری تقویتی، پردازش تصویر، پردازش زبان طبیعی، TensorFlow به یکی از محبوب‌ترین و کاربردی‌ترین فریمورک‌ها در زمینه یادگیری ماشین تبدیل شده است و به کاربران این امکان را می‌دهد تا پروژه‌های مختلف خود را به راحتی پیاده‌سازی کنند.

کتابخانه‌ی PyTorch

پک کتابخانه منبع باز برای یادگیری ماشین و یادگیری عمیق است که توسط فیسبوک توسعه یافته و به ویژه برای پژوهشگران و محققان در حوزه‌های مختلف علمی طراحی شده است. این فریمورک به دلیل سادگی و انعطاف‌پذیری بالایی که در طراحی و آزمایش مدل‌های یادگیری عمیق ارائه می‌دهد، به محبوبیت زیادی دست یافته است. PyTorch از یک ساختار گراف محاسباتی دینامیک (Dynamic Computation Graph) استفاده می‌کند، به این معنی که گراف محاسباتی در زمان اجرا ایجاد می‌شود، که امکان تغییرات سریع و آسان در مدل‌ها را فراهم می‌کند. این ویژگی به پژوهشگران این امکان را می‌دهد که به راحتی آزمایشات مختلف را ایجاد دهن و نتایج را به سرعت مشاهده کنند. علاوه بر سادگی و قابلیت انعطاف‌پذیری، PyTorch ابزارهای متنوعی را برای پردازش داده‌ها، آموزش مدل‌ها و پیوندهای آن‌ها ارائه می‌دهد. این فریمورک دارای یک جامعه کاربری فعال و غنی از مستندات است که به کاربران جدید و باتجریه کمک می‌کند تا از قابلیت‌های آن به بهترین نحو استفاده کنند. PyTorch به ویژه در تحقیق و توسعه مدل‌های پیچیده، از جمله شبکه‌های عصبی کائولوشنی و شبکه‌های عصبی بازگشتی، بسیار مورد استفاده قرار می‌گیرد و به عنوان یک ابزار اصلی در بسیاری از پژوههای علمی و صنعتی شناخته می‌شود.

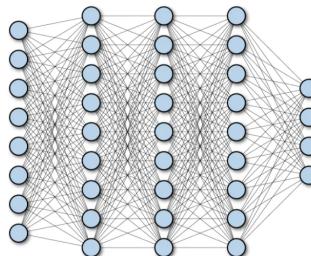
تسک اول

پیاده‌سازی شبکه عصبی fully connected CNN با استفاده از کتابخانه TensorFlow برای دیتاست MNIST

در ابتدا به توضیحی درمورد شبکه‌ی fully connected CNN می‌پردازیم.

CNN

شبکه عصبی کانولوشنی (CNN) یک نوع شبکه عصبی مصنوعی است که به طور ویژه برای پردازش داده‌های دارای ساختار شبکه‌ای مانند تصاویر طراحی شده است. این نوع شبکه‌ها از لایه‌های کانولوشنی استفاده می‌کنند که وظیفه استخراج ویژگی‌های محلی از داده‌های ورودی را دارند. در CNN ها، لایه‌های کانولوشن و تجمعی (Pooling) به طور متنابض قرار می‌گیرند تا ابعاد داده‌ها کاهش یابد و ویژگی‌های مهم از آن‌ها استخراج شود. با این حال، در بخش پایانی یک شبکه CNN، لایه‌های کاملاً متصل (Fully Connected) قرار می‌گیرند که نقش مهمی در تضمین گیری نهایی دارند. این لایه‌ها، برخلاف لایه‌های کانولوشنی که فقط با بخش‌های محلی داده ارتباط دارند، به تمامی نورون‌های لایه قبلی متصل هستند. این لایه‌ها ویژگی‌های استخراج شده توسط لایه‌های کانولوشنی را به صورت یک بردار مسطح درآورده و از این اطلاعات برای کلاس‌بندی یا پیش‌بینی خروجی نهایی استفاده می‌کنند. در واقع، لایه‌های کاملاً متصل شیوه‌ی به شبکه‌های عصبی چندلایه کلاسیک (MLP) عمل می‌کنند که هر نورون آن با تمام نورون‌های لایه قبل ارتباط دارد. این ویژگی باعث می‌شود تا CNN بتواند از قدرت بالای لایه‌های کانولوشنی برای استخراج ویژگی‌ها و از انعطاف‌پذیری و قدرت تضمین گیری لایه‌های کاملاً متصل برای پیش‌بینی خروجی استفاده کند. به عبارت دیگر، لایه‌های کاملاً متصل نقش ترکیب و تجمعی ویژگی‌های سطح بالا را ایفا می‌کنند که در مراحل قبلی توسط لایه‌های کانولوشنی و تجمعی استخراج شده‌اند. لایه‌های کاملاً متصل به دلیل تعداد بالای ارتباطات و وزن‌ها معمولاً بیشترین محاسبات را در CNN به خود اختصاص می‌دهند و بهینه‌سازی این لایه‌ها می‌تواند تأثیر چشمگیری در عملکرد شبکه داشته باشد. در بسیاری از عمارتی‌های مدرن، ویژه برای کاربردهای مرتبط با تصاویر بزرگ و پیچیده، تلاش می‌شود تا تعادلی بین استفاده از لایه‌های کانولوشنی و کاملاً متصل برقرار شود تا هم دقت مدل افزایش یابد و هم زمان محاسباتی کاهش پیدا کند.



توضیح کد پیاده‌سازی شده با استفاده از کتابخانه TensorFlow

برای پیاده‌سازی این شبکه روی دیتاست MNIST ابتدا با استفاده از کتابخانه tensorflow.keras دیتاست را لود می‌کیم، سپس باید داده‌ها را نرمال کنیم، تقسیم کردن پیکسل‌های تصاویر بر ۲۵۵ در این کد به همین منظور انجام می‌شود. در تصاویر دیجیتال، هر پیکسل یک مقدار عددی بین ۰ تا ۲۵۵ دارد که شدت روشنایی آن پیکسل را نشان می‌دهد؛ به معنای سیاه مطلق و ۲۵۵ به معنای سفید مطلق است. با تقسیم کردن مقدار هر پیکسل بر ۲۵۵، این مقادیر به محدوده‌ای بین ۰ و ۱ تبدیل می‌شوند. نرمال‌سازی به چند دلیل مهم انجام می‌شود؛ مثلاً این کار باعث می‌شود تا آموزش شبکه راحت‌تر صورت بگیرد. این امر به این دلیل است که داده‌های نرمال‌شده مقایسه‌یکنواخت دارند و باعث می‌شود که بهینه‌سازی مدل (مانند به وزرسانی وزن‌ها) کارآمدتر انجام شود. علاوه بر آن نرمال‌سازی داده‌ها می‌تواند به غلوبگیری از مشکلاتی مانند overflow یا underflow در طی محاسبات کمک کند، زیرا مقادیر ورودی به شبکه در محدوده‌ای کوچک و کنترل شده قرار می‌گیرند. و در آخر می‌توان گفت که به طور کلی، این روش نرمال‌سازی به بهبود عملکرد مدل‌های یادگیری عمیق کمک می‌کند و به یکی از مراحل رایج پیش‌پردازش داده‌ها تبدیل شده است.

```
train_images, test_images = train_images / 255.0, test_images / 255.0
```

در ادامه، تصاویر دیتاست را reshape می‌کنیم. در دیتاست اولیه، هر تصویر به صورت یک آرایه دو بعدی با ابعاد ۲۸ در ۲۸ ذخیره شده است، که نمایانگر ۲۸ پیکسل در عرض و ۲۸ پیکسل در طول تصویر است. این آرایه دو بعدی به عنوان یک تصویر تک کاتالی (سیاه و سفید) در نظر گرفته می‌شود. برای استفاده از آن‌ها در CNN نیاز است که داده‌های ورودی به شکل خاصی باشند تا شبکه بتواند آنها را پردازش کند. به همین دلیل، در اینجا ابعاد تصاویر از (۲۸،۲۸) به (۱،۲۸،۲۸) تغییر می‌کند. این تغییر شکل به این معناست که هر تصویر ۲۸ در ۲۸ دارای یک کاتال است. که همان کاتال خاکستری یا تک کاتالی برای تصاویر سیاه و سفید است. پس در اینجا تصاویر از حالت ۲ بعدی به حالت ۳ بعدی تبدیل می‌شوند.

```
train_images = train_images.reshape((train_images.shape[0], 28, 28, 1))
test_images = test_images.reshape((test_images.shape[0], 28, 28, 1))
```

بخش بعدی مربوط به تعریف مدل با استفاده از کتابخانه Keras است. این بخش را کمی دقیق‌تر مورد بررسی قرار می‌دهیم. model.sequential() یک مدل خطی از لایه‌ها را تعریف می‌کند که به صورت ترتیبی از اولین لایه به آخرین لایه عبور می‌کند.

ابتدا conv2D یک لایه کاتولوشنی با ۳۲ فیلتر ۳ در ۳ تعریف می‌کند. این لایه ویژگی‌های محلی از تصاویر ورودی را استخراج می‌کند. همچنین در این بخش مشخص می‌شود که ورودی‌های این لایه تصاویری با ابعاد ۲۸ در ۲۸ و یک کاتال (تصاویر سیاه و سفید) هستند. در این کد نهایی، از تابع فعال‌سازی LeakyRelu استفاده شده است که نسخه تغییر یافتهReLU است. در LeReLU . برخلاف راهنمای نیز اجزه عبور می‌یابند، هر چند با شبیه‌ضعیفی برابر با alpha . (در ادامه به توضیح بیشتری از این دو تابع خواهیم پرداخت). لایه MaxPooling ابعاد ویژگی‌ها را کاهش می‌دهد و با انتخاب بیشینه مقدار در هر فیلتر، ابعاد مکانی تصویر را به نصف کاهش می‌دهد. در ادامه تعداد فیلترها در هر لایه کاتولوشنی بعدی افزایش یافته (از ۶۴ به ۱۲۸). که به مدل امکان می‌دهد تا ویژگی‌های پیچیده‌تری را استخراج کند.

لایه Flatten آرایه سه بعدی خروجی از لایه‌های کاتولوشنی را به یک بردار یک بعدی تبدیل می‌کند تا بتوان آن را به لایه‌های Dense متصل کرد. لایه‌های Dense ۶۴ و ۱۲۸ در خط بعدی، دو لایه کاملاً متصل با ۶۴ و ۱۲۸ نورون هستند که به ترکیب و تجمعی ویژگی‌های استخراج شده توسعه لایه‌های قبلی می‌پردازند. (در این لایه‌ها نیز تابع فعال‌سازی LeakyReLU به کار گرفته شده) و در نهایت لایه‌ی خروجی با ۱۰ نورون است که هر نورون نماینده یکی از کلاس‌های ارقام ۰ تا ۹ است.

در بخش کامپایل مدل، از rmsprop برای بهینه‌سازی مدل استفاده می‌شود، که یک نسخه بهبود یافته از روش گرادیان نزولی است. همچنین تابع هزینه انتخاب شده Sparse Categorical Crossentropy است که برای مسائل طبقه‌بندی چندکلاسه مناسب است. برای ارزیابی از معیار accuracy می‌خواهیم استفاده کنیم که آن را هم در این بخش مشخص می‌کیم.

طی تست‌های مختلفی که انجام می‌دهیم و طبق نتایجی که مشاهده می‌کنیم، می‌توانیم تعداد لایه‌ها را هم کم یا زیاد کیم و تاثیر آن را هم در نتایج ببینیم.

```
model = models.Sequential([
    layers.Conv2D(32, (3, 3), input_shape=(28, 28, 1)),
    layers.LeakyReLU(alpha=0.01),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3)),
    layers.LeakyReLU(alpha=0.01),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3)),
    layers.LeakyReLU(alpha=0.01),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128),
    layers.LeakyReLU(alpha=0.01),
    layers.Dense(64),
    layers.LeakyReLU(alpha=0.01),
    layers.Dense(10)
])

model.compile(optimizer='rmsprop',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

در ادامه در هر اپیک فرایند آموزش را انجام می‌دهیم. در نهایت مدل نهایی را برای داده‌های تست اجرا کرده و نتایج را می‌بینیم. طی این فرایند، تعداد epoch‌ها را می‌توانیم تغییر دهیم و تاثیر آن را مشاهده کنیم. علاوه بر آن، از activation function optimizer‌ها و های متفاوتی می‌توانیم استفاده کرده و نتایج را با هم مقایسه کنیم. در ادامه به توضیح مختصاتی از این توابع ارائه می‌شود. همچنین برخی نتایج هم قرار داده خواهد شد؛ مشاهده می‌شود که در هیچ کدام از حالت‌ها دچار overfitting یا underfitting نشده‌ایم.

توابع فعالسازی Relu و Leaky Relu

ReLU و Leaky ReLU هر دو تابع فعالسازی در شبکه‌های عصبی هستند که برای معرفی غیرخطی بودن به مدل استفاده می‌شوند. ReLU مقدار ورودی‌های منفی را به صفر تبدیل می‌کند و ورودی‌های مثبت را بدون تغییر عبور می‌دهد. این ویژگی باعث سرعت بالا و سادگی در محاسبات می‌شود، اما مشکل اصلی ReLU این است که اگر نورون‌ها به منطقه منفی وارد شوند (یعنی وزن‌ها و بایاس‌ها به گونه‌ای تنظیم شوند که ورودی‌های منفی تولید شود)، آن نورون‌ها ممکن است به طور دائمی غیرفعال شوند، که به این وضعیت "مرگ" ReLU گفته می‌شود. Leaky ReLU این مشکل را با اجازه دادن به یک شبکه کوچک (عموماً یک مقدار کوچک مثل $0.01 \dots 0.001$) برای ورودی‌های منفی حل می‌کند؛ به این ترتیب، حتی برای مقداری منفی نیز مقداری گرادیان وجود دارد که باعث می‌شود نورون‌ها کمتر به وضعیت "مرگ" برسند و بهبود یادگیری کمک می‌کند.

الگوریتم‌های بهینه‌سازی Adam و RMSprop

هر دو الگوریتم‌های بهینه‌سازی در یادگیری عمیق هستند که برای بهبود سرعت و کیفیت یادگیری در شبکه‌های عصبی طراحی شده‌اند.

RMSprop (Root Mean Square Propagation)

به طور خاص برای مدیریت نوسانات گرادیان در هنگام یادگیری طراحی شده است. این الگوریتم با استفاده از میانگین مربع گرادیان‌های گذشته، یک نرخ یادگیری متناسب با هر پارامتر تولید می‌کند. به عبارت دیگر، RMSprop نرخ یادگیری را برای هر پارامتر بدطور جداگانه تنظیم می‌کند و به پارامترهایی که گرادیان‌های بزرگ‌تری دارند، نرخ یادگیری کمتری می‌دهد. این ویژگی به جلوگیری از نوسانات شدید در بهروزرسانی وزن‌ها کمک می‌کند.

Adam (Adaptive Moment Estimation)

ترکیبی از دو الگوریتم RMSprop و Momentum است. این الگوریتم نه تنها از میانگین مربع گرادیان‌های گذشته استفاده می‌کند، بلکه میانگین متحرک گرادیان‌ها را نیز در نظر می‌گیرد. به این ترتیب، Adam از اطلاعات بیشتری درباره رفتار گرادیان‌ها در طول زمان استفاده می‌کند تا نرخ یادگیری را بهینه کند. Adam عموماً برای اکثر مسائل یادگیری عمیق عملکرد بهتری دارد و به دلیل تنظیمات خودکار نرخ یادگیری، به طور گستره‌های مورد استفاده قرار می‌گیرد.

به طور خلاصه، در حالی که RMSprop بر پایه میانگین مربع گرادیان‌ها عمل می‌کند، Adam به دنبال استفاده از هر دو میانگین مربع و میانگین متحرک گرادیان‌هاست که منجر به بهینه‌سازی بهتری در اکثر شرایط می‌شود.

نتایج تسک اول

(دو نتیجه‌ی بهتر با توجه به نمودارها، مشخص شده‌اند.)

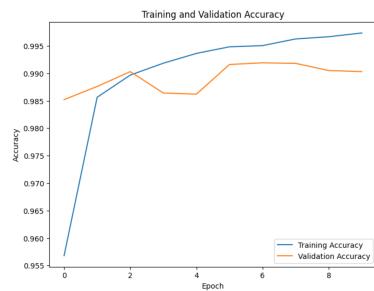
activation function: Relu

optimizer: Adam

total layers: 8

number of epochs: 10

(test accuracy: 0.9902999997138977)



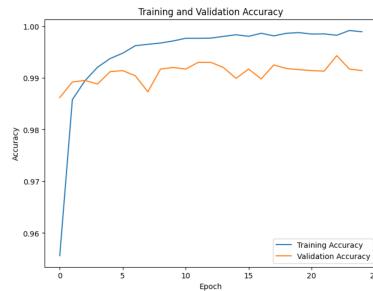
activation function: Relu

optimizer: Adam

total layers: 8

number of epochs: 25

(test accuracy: 0.9914000034332275)



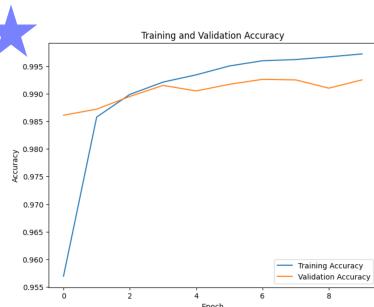
activation function: Leaky Relu

optimizer :Adam

total layers: 12

number of epochs: 10

(test accuracy: 0.9925000071525574)



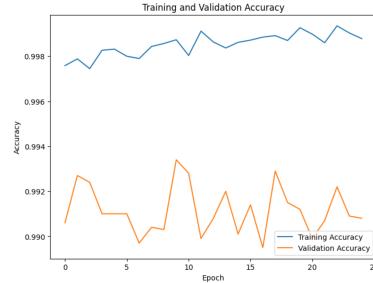
activation function: Leaky Relu

optimizer: Adam

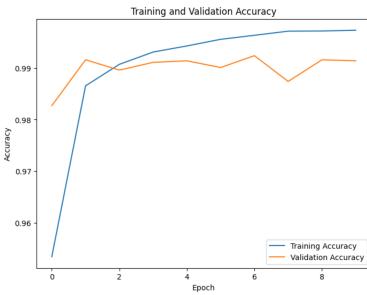
total layers: 12

number of epochs: 25

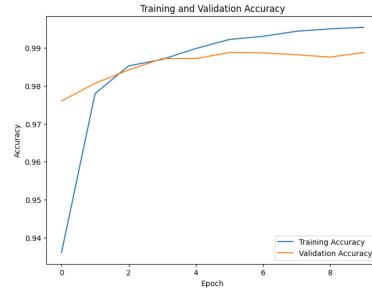
(test accuracy: 0.991400003433275)



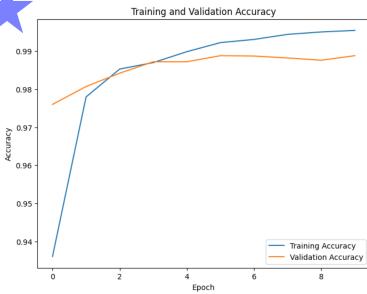
activation function: Leaky Relu
optimizer: Adam
total layers: 12
number of epochs: 10
(test accuracy: 0.9914000034332275)



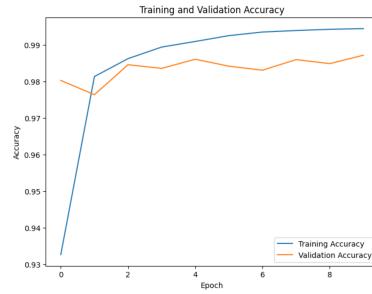
activation function: Leaky Relu
optimizer: rmsprop
total layers: 12
number of epochs: 10
(test accuracy: 0.9887999892234802)



activation function: Leaky Relu
optimizer: rmsprop
total layers: 15
number of epochs: 10
(test accuracy: 0.9887999892234802)



activation function: Leaky Relu
optimizer: rmsprop
total layers: 12
number of epochs: 10
(test accuracy: 0.9872000217437744)



تجربیات تسک اول

با توجه به این که با شبکه CNN از قبل آشنایی داشتم، در این تسک بیشتر با کتابخانه TensorFlow آشنا شدم. همچنین از آنجایی که نمایش نمودارها در مدل‌های هوش مصنوعی بسیار مهم و کاربردی است، کمی بیشتر متدهای نمایش نمودارها در کتابخانه matplotlib را یاد گرفتم، در این تسک، نتایج را با optimizer‌ها و activation function‌ها و همچنین با epoch‌ها و batchsize‌ها متفاوتی تست کردم و بعضی از تفاوت‌های آن‌ها و تاثیرشان آشنا شدم.

تسک دوم

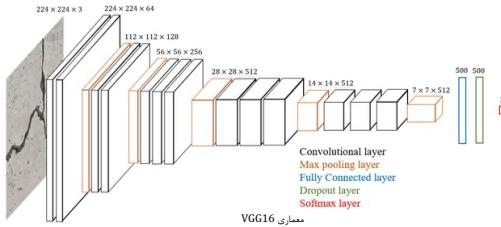
پیاده‌سازی شبکه VGG با استفاده از کتابخانه PyTorch برای دیتاست CIFAR-10

VGG

شبکه VGG (Visual Geometry Group) یکی از معماری‌های پیشرفته و مؤثر در زمینه شبکه‌های عصبی کانولوشنی است که توسط محققان دانشگاه آکسفورد در سال ۲۰۱۴ معرفی شد. این شبکه به دلیل ساختار ساده و منظم خود که از لایه‌های کانولوشنی با اندازه ثابت ۳ در ۳ و لایه‌های Max Pooling ۲x2 تشکیل شده، شناخته می‌شود. یکی از ویژگی‌های بارز VGG این است که عمق شبکه افزایش یافته و تعداد لایه‌های کانولوشنی به ۱۶ یا ۱۹ لایه می‌رسد. این عمق به شبکه این امکان را می‌دهد تا ویژگی‌های پیچیده‌تری را از تصاویر استخراج کند و دقت بالاتری در طبقبندی تصاویر به دست آورد.

در دو نسخه اصلی، VGG16 و VGG19، معرفی شده است که به ترتیب به ۱۶ و ۱۹ لایه قابل آموزش اشاره دارند. یکی از نکات قوت VGG قابلیت انتقال یادگیری (Transfer Learning) است، به این معنی که می‌توان از وزن‌های آموزش دیده این مدل برای حل مسائل مختلف استفاده کرد و آن را به سادگی برای داده‌های جدید تنظیم کرد.

با این حال، یکی از معایب VGG، نیاز به منابع محاسباتی بالا و زمان آموزش طولانی است. تعداد زیاد پارامترها در این شبکه باعث می‌شود که به حافظه و قدرت پردازشی بیشتری نیاز داشته باشد. به همین دلیل، در سال‌های اخیر، مدل‌های جدیدتر و کارآمدتری مانند Inception و ResNet طراحی شده‌اند که تلاش می‌کنند تا به همین دقت با پیچیدگی کمتری دست یابند.



مقایسه CNN و VGG

در حالی که VGG یک نوع خاص از شبکه‌های عصبی کانولوشنی (CNN) است، تفاوت‌های عمده‌ای بین آن و CNN‌های سنتی وجود دارد. VGG با استفاده از لایه‌های کانولوشنی عمیق و ثابت (۳ در ۳) به همراه لایه‌های Max Pooling قادر است ویژگی‌های پیچیده‌تری را از تصاویر استخراج کند و در نتیجه دقت بالاتری در طبقبندی ارائه دهد. در مقابل، CNN‌های ساده‌تر ممکن است از لایه‌های کانولوشنی با اندازه‌های مختلف و عمق کمتری استفاده کنند، که این می‌تواند منجر به کاهش دقت شود. به طور کلی، VGG به عنوان یک مدل عمیق‌تر و پیچیده‌تر در مقایسه با CNN‌های معمولی، نیاز به منابع بیشتری دارد و برای کاربردهای خاص و چالش‌برانگیز مناسب‌تر است.

توضیح کد پیاده‌سازی شده با استفاده از کتابخانه PyTorch

بخش سمت کردن certificate برای این است که به راحتی بتوانیم دیتابست CIFAR10 را دانلود کنیم.

پیش از آن که دیتابست را لود کنیم، بخش transforms ها را برای دیتابی تست و تربین داریم .transforms.compose این امکان را به ما می‌دهد که چند تابع مختلف را بصورت زنجیره‌ای ترکیب کنیم، ما از توابع مختلفی در آن استفاده می‌کنیم و همه را یک ماره روی داده‌های ورودی هر بخش اعمال می‌کنیم. سیاری از این پیش‌پردازش‌ها در جهت کم کردن اختلال رخداد استفاده می‌شوند و همچنین کمک می‌کند تا مدل به بهترین نحو ممکن یاد بگیرد و دقت بالاتری را در عملکرد overfitting خود ارائه دهد.

تصویر ۴ پیکسل حاشیه اضافی اضافه می‌شود. این کار به مدل کمک می‌کند تا یاد بگیرد ویژگی‌ها را از قسمت‌های مختلف تصویر استخراج کند و به طور کلی به افزایش تعمیم‌پذیری مدل کمک می‌کند.

کار به افزایش تنوع داده‌های آموزشی کمک می‌کند و به مدل می‌آموزد که از ویژگی‌های مختلف و جهات مختلف تصاویر بهره‌برداری کند transforms.RandomHorizontalFlip یک تبدیل تصادفی است که تصویر را به صورت افقی ممکوس می‌کند. این

به صورت عددی در بازه ۰ تا ۲۵۵ هستند، به مقادیر نرمالیزه شده در بازه ۰ تا ۱ تبدیل می‌شوند. این تبدیل به شبکه کمک می‌کند تا به طور مؤثرتری با داده‌ها کار کند و همچنین برای پردازش در مدل‌های یادگیری عمیق لازم است. transforms.Normalize تصویر را نرمالیزه می‌کند. این تابع به هر کانال رنگ (RGB) مقادیر میانگین و انحراف معیار مشخصی را اختصاص می‌دهد. در اینجا، میانگین (۰.۴۹۱۴، ۰.۴۸۲۲، ۰.۴۴۶۵) و انحراف معیار برای هر کانال (۰.۲۰۳۳، ۰.۱۹۹۴، ۰.۲۰۱۰) هستند.

این مقادیر، مقادیر استاندارد هستند. نرم‌افزاری به کاهش تأثیر نوسانات بین داده‌ها کمک می‌کند و به مدل کمک می‌کند تا یاد بگیرد که ویژگی‌های مهم را از تصاویر شناسایی کند. این نکته را هم باید مدنظر داشته باشیم که روی داده‌های تست، معمولاً فقط نرمالیزیشن انجام می‌شود و از موارد دیگر استفاده نمی‌شود تا ارزیابی دقیقاً روی داده‌های تست صورت بگیرد.

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

طبق موارد مدنظر، ۱۰ درصد از دیتابی تستین را به عنوان دیتابی validation در نظر می‌گیریم تا در طی آموزش مدل از آن استفاده کنیم. سپس طبق معماری VGG16، لایه‌ها را تعریف می‌کنیم.

در فرایند آموزش مدل، هایپر پارامترهای مختلفی وجود دارد که باید آن‌ها را مشخص کنیم. برای مثال، batch size در number of epochs ... همچنین از معیارها و بهینه‌سازهای متفاوتی می‌توان استفاده کرد؛ با این حال بعضی از آن‌ها بهتر عمل می‌کنند و معمول‌تر هستند. در این کد از criterion برای CrossEntropy و همچنین از SGD یا همان گرادیان نزولی تصادفی به عنوان optimizer استفاده شده lr. همان نرخ یادگیری است که یکی دیگر از هایپر پارامترهای است. این مقدار تعیین می‌کند که وزن‌ها چقدر آپدیت شوند. اگر مقدار آن خلی کم تغیین شود فرایند آموزش خیلی کند می‌شود و اگر خیلی زیاد باشد ممکن است در مینیمم‌های محلی گیر کنیم و هیچ گاه به بهینه‌ترین حالت نرسیم momentum. یک تکنیک است

که به SGD اضافه می شود تا سرعت همگرایی را افزایش دهد و از نوسانات شدید در بروزرسانی وزن ها جلوگیری کند. این مقدار تعیین شده؛ یعنی تاثیر گرادیان های قلی در به روزرسانی ها، ۹۰ درصد است. میزان weight decay مشخص می کند که میزان جریمه وزن های بزرگ چقدر است. این تکیک برای کاهش overfitting استفاده می شود.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = VGG().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
```

حالا ابتدا بخش آموزش و سپس تست را الجام می دهیم. توابع train و evaluate به همین منظور پیاده سازی شده. در نهایت

برای هر اپیک test accuracy و validation accuracy را به همراه نمودارهای مربوطه نمایش می دهیم یک نکته‌ی قابل توجه دیگر این است در پیاده سازی مدل‌ها، ممکن است با مسائل زیادی رو به رو شویم؛ به همین دلیل بهتر است مدل را ذخیره کنیم. در این کد هم مدل‌های هر اپیک را ذخیره می‌شوند. بهترین مدل هم به صورت جداگانه سیو می‌شود تا در ادامه از آن استفاده کنیم. علاوه، کافیگ‌های مربوطه را هم می‌توانیم سیو کنیم.

```
config = {
    "batch_size": 128,
    "learning_rate": 0.01,
    "epochs": 50,
    "transform_train": str(transform_train),
    "transform_test": str(transform_test),
    "device": str(device)
}
with open('./model_checkpoints/config.json', 'w') as f:
    json.dump(config, f, indent=4)

train_losses = []
train_accuracies = []
val_losses = []
val_accuracies = []
best_val_accuracy = 0

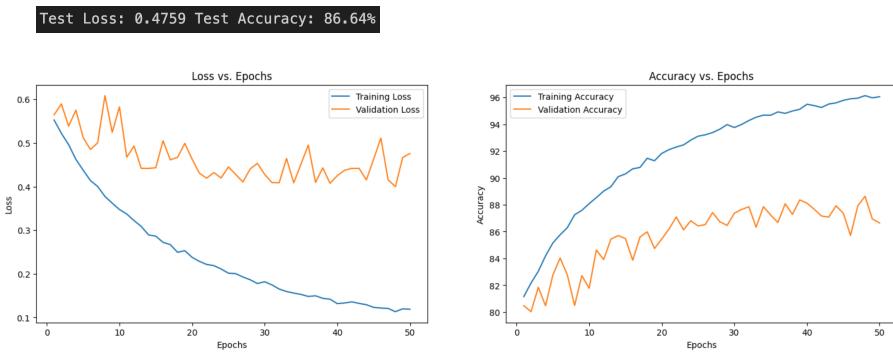
for epoch in range(config['epochs']):
    train_loss, train_accuracy = train(model, trainloader, criterion, optimizer, epoch, device)
    val_loss, val_accuracy = evaluate(model, testloader, criterion, device)

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

    print(f'Epoch [{epoch+1}/{config["epochs"]}]:')
    print(f'Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.2f}%')
    print(f'Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_accuracy:.2f}%')

    torch.save(model.state_dict(), f'./model_checkpoints/model_{epoch}_{epoch+1}.pth') ← ذخیره مدل‌ها در هر epoch
    if val_accuracy > best_val_accuracy:
        best_val_accuracy = val_accuracy
        torch.save(model.state_dict(), './model_checkpoints/best_model.pth') ← ذخیره بهترین مدل
```

نتایج تسک دوم (بهترین نتیجه)



تجربیات تسک دوم

در این تسک نسبت به تسک قبلی با موارد بیشتری سروکار داشتم، با شبکه‌ی VGG آشنایی نداشتم و ابتدا درمورد آن مطالعه کردم. قرار بر این بود که این مدل با استفاده از کتابخانه PyTorch پیاده‌سازی شود؛ به همین دلیل سعی کردم متدهای موردنیاز در آن را بیشتر یاد بگیرم. با متدهای مختلفی جهت کاهش overfitting مانند RandomSizeCrop و RandomRotation ... آشنا شدم. متوجه شدم که تمام این‌ها را می‌توان با استفاده از متدهای transform در کنار هم تعریف کرده و سپس همه را با هم روی دیتا اعمال کرد. همان طور که در توضیحات قبل تر اشاره شد، ذخیره‌ی مدل‌ها مساله‌ی مهمی است. در این تسک شیوه‌ی ذخیره‌ی مدل و کانفیگ‌ها را هم یاد گرفتم. با توجه به این که در این تسک خواسته شد از دیتای ولیدیشن هم در کنار دیتای تست و ترین استفاده شود و پیش از این با استفاده از آن آشنایی نداشتم، این مورد هم یکی از مسائلی بود که طی انجام این تسک بهتر یاد گرفتم.

تسک سوم

پیاده‌سازی شبکه ResNet با استفاده از کتابخانه PyTorch برای دیتاست CIFAR-10

ResNet

یک نوع معماري شبکه عصبی عميق است که توسط محققان مایکروسافت در سال ۲۰۱۵ معرفی شد. به ویژه برای حل مشکل "تضعيف گراديان" که در شبکه‌های بسيار عميق رخ می‌دهد، طراحی شده است.

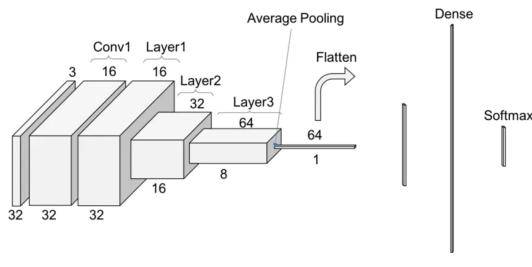
در شبکه‌های عصبی عميق، زمانی که لايه‌های زبادي به مدل اضافه می‌شوند، گراديان‌های محاسبه شده در طول فرآيند یادگيری ممکن است بسيار كوچك شوند، به طوري که وزن‌های لايه‌های اوليه شبکه به درستي به روزرسانی نمي‌شوند. اين مسئله باعث می‌شود که دقت شبکه با افزایش تعداد لايه‌ها کاهش پابد، که به آن "تضعيف گراديان" گفته می‌شود. ResNet اين مشكل را با معرفی "انصالات باقی‌مانده" (Residual Connections) یا "پيوندهای پرشی" (Skip Connections) "حل می‌کند.

در حقیقت در ResNet، به جای اینکه هر لايه به صورت مستقيم به لايه بعدی متصل شود، خروجي برخی از لايه‌ها مستقيماً به لايه‌های بعد از آن اضافه می‌شود. اين عمليات به صورت زير انجام می‌شود:

$$y = F(x) + x$$

معادله $y = F(x) + x$ را در نظر می‌گيريم. در اين معادله، x ورودي لايه، y خروجي لايه فعلي به عناء ورودي اوليه (با استفاده از اتصال باقی‌مانده) و $F(x)$. تابعی است که توسط چندين لایه (مثلًا ترکيبی از لايه‌های کاتولوشن و فعال‌سازی) ايجاد می‌شود. اين ساختار باعث می‌شود که مدل سريع‌تر به همگاري برسد و همچنين به یادگيری بهينه‌تر هم كمک می‌کند. پس همان طور که اشاره كردیم، اين معماري امكان ساخت مدل‌های بسيار عميق (تا صدها لايه) را بدون مواجهه با مشكل تضعيف گراديان فراهم می‌کند. نسخه‌های مختلفی از ResNet20 مانند... ResNet18, 34, 50, ... وجود دارد که اين اعداد تعداد لايه‌ها را نشان می‌دهند.

علاوه بر معماري گفته شده، نوع دیگري از آن هم به عنوان يكی از نسخه‌های ساده‌تر و سبک‌تر ResNet20 به نام ResNet20 معرفی شد تا به طور خاص برای مجموعه داده‌های كوچکتر و ساده‌تری CIFAR-10 بهينه‌سازی شود. به طور خلاصه از دلایل اصلی معرفی آن می‌توانيم به مواردي مانند انبطاق با دیتاست‌های كوچکتری مانند CIFAR-10 آموزش سريع و منابع محاسباتی كمتر و يك معماري پايه‌اي برای اين نوع معماري در کنار دارا بودن مزایای انصالات بیان شده، اشاره کرد.



توضیح کد پیاده‌سازی شده با استفاده از کتابخانه PyTorch

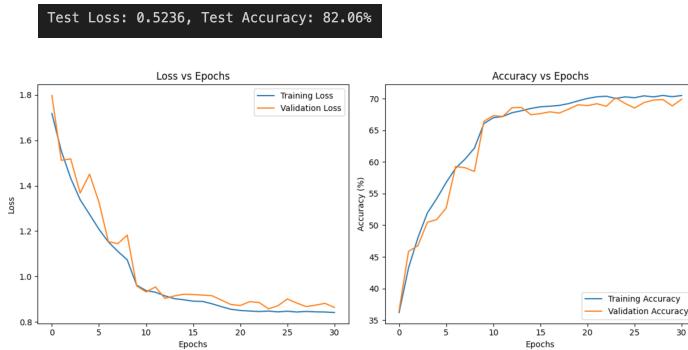
کد پیاده‌سازی شده برای این بخش، بسیار شبیه به کد تسک دوم است و تفاوت اصلی آن تنها در بخش تعریف مدل است. برای انجام این تسک ابتدا شبکه ResNet18 را پیاده‌سازی کردم. سپس برای بررسی این که آبا ResNet20 بهتر عمل می‌کند یا نه، آن را هم تست کردم. نتیجه‌ی هر دو به هم نزدیک بود، با این حال بنظر می‌رسد ResNet18 تا حدی بهتر عمل کرده. در تعریف مدل برای ResNet18، تمام لایه‌ها به صورت صریح تعریف شدند ولی در ResNet20 از پیاده‌سازی ماژولارتی استفاده کردم. همچنین برای بهتر شدن عملکرد مدل، ColorJitter و RandomRotaion را هم در بخش ترنسفورم دیتای تربین اضافه کردم. RandomRotation تصاویر را به صورت تصادفی به اندازه‌ی الفا تا منفی الفا می‌چرخاند (در اینجا زاویه را ۱۵ تعریف کرده‌ایم). این کار باعث می‌شود مدل به جای تمرکز روی جایگیری اشیا در تصویر، به ویژگی‌های مهمتر آن‌ها اهمیت بدهد. همچنین ColorJitter روشانی، کنتراست و اشباع رنگ را به صورت تصادفی تغییر می‌دهد. میزان بیان شده برای هر یک نشان‌دهنده‌ی این است که تا چه مقدار می‌تواند تغییر کند. این کار باعث می‌شود حساسیت مدل در شرایطی که تصاویر در دنیای واقعی گرفته شده‌اند و نور آن‌ها در موارد بیان شده متفاوت است، زیاد نباشد و روی ویژگی‌های مهمتر تصویر تمرکز کند.

```
transform_train = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

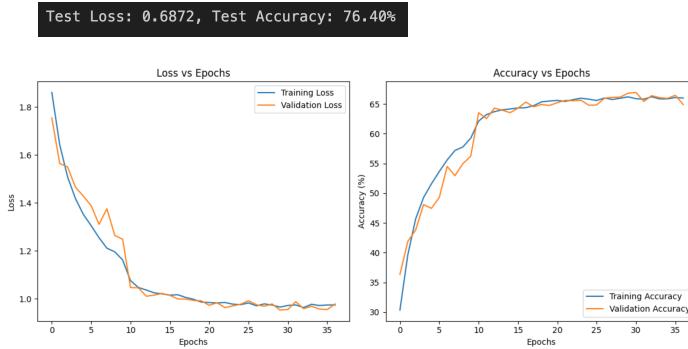
```

نتایج تسک سوم

ResNet18



ResNet20



تجربیات تسک سوم

تسک سوم نسبت به تسک قبلی چالشی تر بود. در تسک قبلی نتیجه‌ی دلخواه سریع‌تر به دست آمد؛ ولی در این تسک علاوه بر این که باید با معماری شبکه‌های ResNet آشنا می‌شدم، نتیجه‌ی خوبی از پیاده‌سازی نمی‌گرفتم و این مساله مقداری زمان بردا. این موضوع باعث شد از data augmentation های بیشتری استفاده کنم. همچنین متوجه شدم که برای مشاهده‌ی ساختار مدل و پارامترهایی که پیاده‌سازی و استفاده کردم، می‌توانم کتابخانه‌های summary و torchvision را به کار ببرم.

تسک چهارم

اعمال Backdoor Attack بر روی شبکه‌ی VGG16 تعریف شده در تسک دوم

Backdoor Attack

حملات backdoor یکی از تهدیدات جدی و پیشرفتنه در حوزه امنیت مدل‌های یادگیری عمیق هستند. در این نوع حملات، مهاجم یک الگوی مخفی یا "تریگر (trigger)" را در مدل طی مرحله آموزش تعییه می‌کند، بهطوری که مدل در شرایط عادی به درستی عمل می‌کند، اما هنگامی که ورودی‌های خاصی با این تریگر به مدل داده می‌شوند، خروجی نادرستی تولید می‌کند. این تریگر می‌تواند هر چیزی باشد، از یک تغییر جزئی در پیکسل‌های تصویر تا یک نشان خاص که به ورودی‌ها افزوده می‌شود. مهاجم برای پیاده‌سازی یک حمله backdoor، باید در مرحله آموزش مدل دخالت داشته باشد. این دخالت ممکن است با تغییر داده‌های آموزشی یا حتی با دستکاری مستقیم پارامترهای مدل صورت گیرد. به عنوان مثال، در یک سیستم تشخیص چهره، مهاجم ممکن است با افروزن یک الگوی خاص، مثل یک برجسب یا شیء کوچک در تصاویر، مدل را به گونه‌ای آموزش دهد که این الگو به عنوان تریگر عمل کند. بعد از اینکه مدل با این داده‌های دستکاری شده آموزش دید، در زمان اجراء، مدل به درستی عمل می‌کند مگر اینکه ورودی خاصی با آن الگوی مخفی (تریگر) ارائه شود. در این حالت، مدل به اشتباه ورودی را طبقه‌بندی می‌کند.

یکی از جالش‌های اصلی در برابر این نوع حملات، شناسایی و تشخیص آنهاست. از آنجایی که تریگر تنها در شرایط خاصی فعال می‌شود و مدل در بقیه موارد به درستی عمل می‌کند، تشخیص حمله به صورت مستقیم و تنها با مشاهده عملکرد کلی مدل دشوار است. این حملات به ویژه زمانی خط‌ناتک می‌شوند که مدل‌ها بر روی داده‌ها پا معماری‌هایی که توسط اشخاص ثالث (third parties) ارائه شده‌اند، آموزش بیینند. در این موارد، اعتماد به یکپارچگی فرآیند آموزش بسیار سخت است و مهاجم می‌تواند به راحتی یک backdoor را بدون آگاهی توسعه‌دهنده تعییه کند.

پیامدهای حملات backdoor می‌توانند بسیار جدی باشند و از دسترسی غیرمجاز به سیستم‌ها، ایجاد اطلاعات نادرست یا حتی ایجاد خرابی در سیستم‌ها منجر شوند. این حملات می‌توانند به طور مستقیم امنیت داده‌ها و حریم خصوصی کاربران را تهدید کنند. به همین دلیل، توسعه روش‌های مؤثر برای شناسایی و کاهش تاثیر این حملات از اهمیت زیادی برخوردار است. روش‌های مختلفی از جمله تحلیل رفتار مدل، تست‌های ورودی-خروجی گسترشده، و استفاده از تکنیک‌های یادگیری ماشینی برای شناسایی الگوهای مشکوک به کار گرفته می‌شوند. با پیشرفت شبکه‌های عصبی عمیق و استفاده گسترده از آنها در برنامه‌های حساس، نیاز به رویکردهای امنیتی پیشرفتنه برای محافظت در برابر حملات backdoor روز به روز بیشتر می‌شود.

توضیح کد پیاده‌سازی شده

در این تسک، هدف اعمال این حمله روی شبکه VGG16 که در تسک دوم پیاده‌سازی شد است. برای انجام آن، ابتدا یک تابع برای تعریف تریگر مدنظر تعریف می‌کنیم. طی انجام این تسک، تریگرهای مختلفی تست شد که در ادامه بعضی از آن‌ها به همراه نتایج مربوطه قرار داده می‌شود. طی اعمال **backdoor attack**، با توجه به هدفی که شخص دارد، تریگرها می‌توانند روی دیتای ولیدیشن اعمال شوند یا نشوند؛ با این حال اکثر اوقات دیتای ولیدیشن را آلووه نمی‌کنند. این تسک هم به همین منوال انجام گرفته، یکی دیگر از مواردی که باید مشخص شود، درصد دیتای آلووه شده است. این فرایند با درصدهای متفاوت هم تست شد و در نهایت fraction ۱۵ درصد در بهترین نتایج استفاده شده است. در تابع بعدی، با توجه به این درصد، بصورت رندوم ایندکس‌های دیتاپوینت‌ها انتخاب شده و تریگر روی آن‌ها اعمال می‌شود. برای دیتای تست و تربین این مراحل را انجام می‌دهیم.

ادامه‌ی کار مانند قبل است و مدل باید آموزش ببیند. برای ارزیابی این که حمله‌ی ما چقدر موثر بوده از معیار bsr یا backdoor success rate استفاده می‌کنیم؛ این معیار مشخص می‌کند خروجی مدل برای چند درصد از دیتا با کلاس هدف مشخص شده مطابقت دارد.

یکی دیگر از نکاتی که طی کد باید به آن توجه شود، تفاوت فرمت تصاویر در numpy و pyTorch است. با توجه به این که تصاویر در pyTorch با فرمت (C, H, W) و در نامهای بصورت (H, W, C) هستند، نیاز است که در مواردی (مانند ابتدای تابع trigger add) بررسی کنیم که اگر فرمات‌ها با هم یکی نیستند، تبدیل مربوطه را انجام دهیم تا به مشکل نخوریم.

```
if isinstance(image, torch.Tensor):
    # Convert to NumPy array with shape (32, 32, 3)
    image_np = np.array(image.permute(1, 2, 0))
else:
    image_np = image
```

این که چه تریگری بهتر عمل می‌کند به دیتاست مدنظر سنتگی دارد؛ مثلاً شاید ب Fletcher باید اگر تریگر یک مریع ۸ در ۸ شطرنجی با در رنگ مختلف باشد بهتر عمل می‌کند، در حالی که ممکن است در عمل واقعاً این طور نباشد و مریع ۸ در ۸ با یک رنگ ثابت بهتر عمل کند. همچنین نتایج برای Hallتی که محل تریگر بصورت رندوم در هر دیتاپوینت عوض شود هم تست شد ولی bsr به درصد بالایی نرسید.

استفاده از دو تریگر به صورت همزمان - یک مریع ب بنفس به همراه خط دور تا دور تصویر به رنگ سبز - نسبت به تست‌های قبلی نتایج بهتری به همراه داشت.

با این حال، بهترین نتایج مربوط به مریع ۸ در ۸ بنفس رنگ بود.

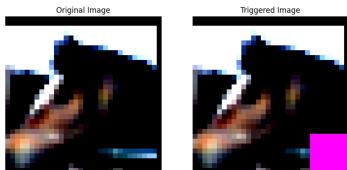
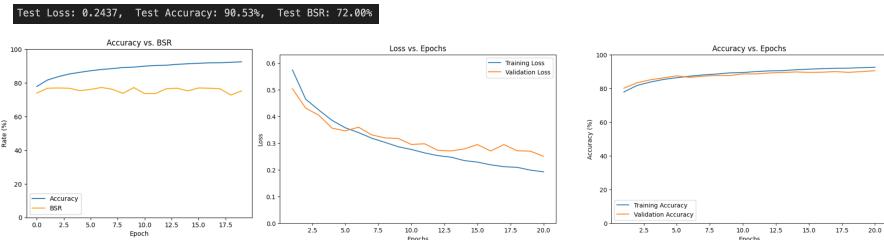
مریع رنگی، نتایج بسیار بهتری نسبت به مریع مشکی داشت. این تفاوت‌ها به نوع تصاویر موجود در دیتاست مربوط است. همچنین این نکته را هم باید در نظر داشته باشیم که هر بار دیتا به صورت رندوم تقسیم می‌شود و این رندوم بودن هم در خوب بودن یا نبودن نتایج موثر است. (پس برای مثال، ممکن است در یک split دیگر از دیتا، حالت ۲ تریگری که نسبت به خیلی از تست‌ها نتایج بهتری داشت، بهتر از بهترین نتایجی که تا به حال گرفته‌ایم عمل کند.)

در همه‌ی تست‌ها کلاس هدف، کلاس "0" انتخاب شد.

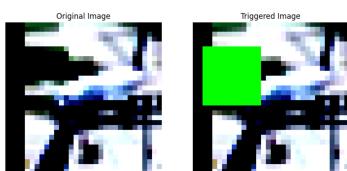
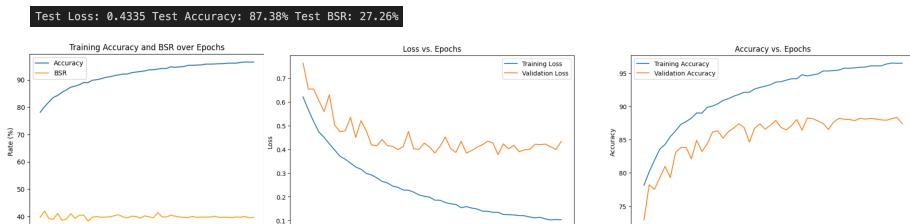
نتایج تsek چهارم

نتایجی که در ادامه قرار داده می شود صرفا برای نشان دادن برخی تست های صورت گرفته است و مشخص است که برای یک مقایسه خوب و اصولی، باید تنها یک پارامتر بین حالت های مختلف فرق داشته باشد.

- مریع ۸ در ۸ بنفش رنگ، $\text{fraction} = 15\%$ (بهترین نتیجه)

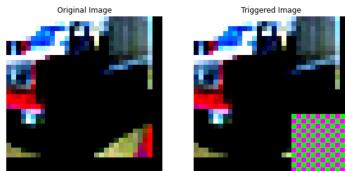
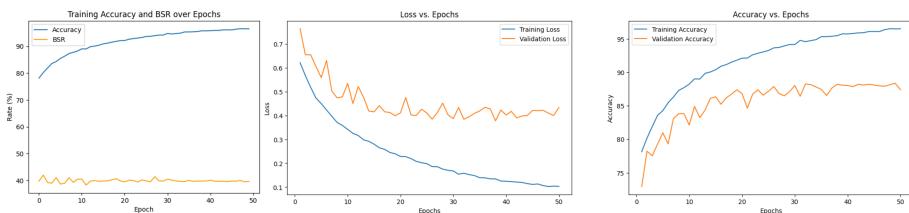


- مریع ۱۲ در ۱۲ سبز رنگ، $\text{fraction} = 10\%$ با جا بهایی رندوم:



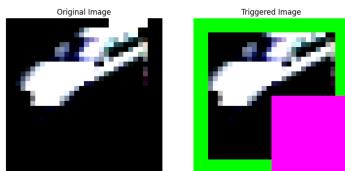
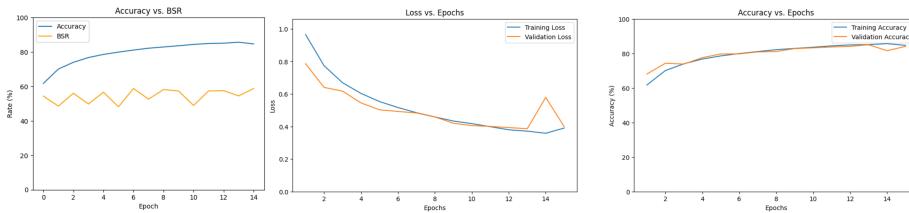
• مربع ۱۲ در ۱۲ شطرنجی بنفس و سبز، fraction = 10%

Test Loss: 0.4335 Test Accuracy: 87.38% Test BSR: 27.26%

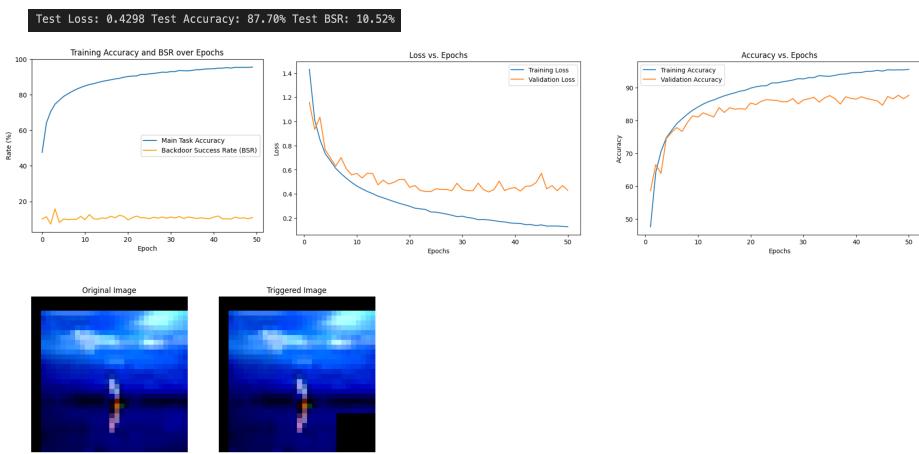


• مربع ۱۶ در ۱۶ بنفس زنگ به همراه خط دور تصویر سبز، fraction = 25%

Test Loss: 0.3667, Test Accuracy: 84.30%, Test BSR: 60.56%



• مربع ۸ در ۸ مشکی، fraction = 1%



تجربیات تسک چهارم

موضوع این تسک نسبت به تسک‌های قبلی جدیدتر بود. از آنجایی که تا به حال با حملات شبکه عصبی آشنا نداشتیم، تسک جالب‌تری بود و مشتاق بودم کارهای بیشتری در این راستا انجام می‌شد. در راستای یاد گرفتن backdoor attack با trigger أشنا شدم و همچنین معیار BSR