



به نام خدا



دانشگاه تهران

دانشکده مهندسی برق و کامپیو تر

شبکه های عصبی و یادگیری عمیق

Mini Project #1

| | |
|---------------------|--------------------|
| پرهام زیلوچیان مقدم | نام و نام خانوادگی |
| 810198304 | شماره دانشجویی |
| 1399/2/3 | تاریخ ارسال گزارش |

فهرست گزارش سوالات (لطفاً پس از تکمیل گزارش، این فهرست را به روز کنید.)

| | |
|---------|---------------------|
| 3..... | سوال 1 – بخش تشریحی |
| 6..... | سوال 2 – بخش تشریحی |
| 7..... | سوال 3 – بخش تشریحی |
| 11..... | سوال 4 – بخش تشریحی |
| 20..... | سوال 5 – بخش تشریحی |
| 22..... | سوال 6 – بخش تشریحی |
| 25..... | بخش عملی |

سوال 1 - بخش تشریحی

تابع هزینه تابعی هست که Performance یک مدل طراحی شده را برای داده‌هایی که به شبکه می‌دهیم، مورد آزمایش قرار می‌دهد. تابع هزینه یا Cost Function خطای میان داده‌های پیش‌بینی شده و داده‌های مورد انتظار را محاسبه می‌کند و سپس مقدار آن را در قالب یک عدد به ما باز می‌گرداند.

حال در این سوال از ما خواسته‌اید که برای مسائل Classification و نیز همین‌طور مسائل Regression دو نمونه تابع هزینه را نام ببریم و سپس عملکرد آن تابع را توضیح دهیم.

Regression Problems -

:MAE - 1

تابع هزینه یا Mean Absolute Error MAE Loss Function می‌باشد، در حقیقت می‌آید و میانگین بزرگی خطاهای را در یک گروه از پیش‌بینی‌های انجام شده محاسبه و اندازه‌گیری می‌کند. به زبان دیگر این روش میانگین قدرمطلق تفاضل‌های میان داده‌های پیش‌بینی شده و مقادیر واقعی آن‌ها در زمانی که تمامی داده‌ها اهمیت برابر دارند، می‌باشد. فرمول آن نیز مطابق زیر می‌باشد:

$$MAE = \frac{1}{m} \sum_{i=1}^m |\hat{y}^{(i)} - y^{(i)}|$$

شکل 1: فرمول محاسبه MAE

:MSE - 2

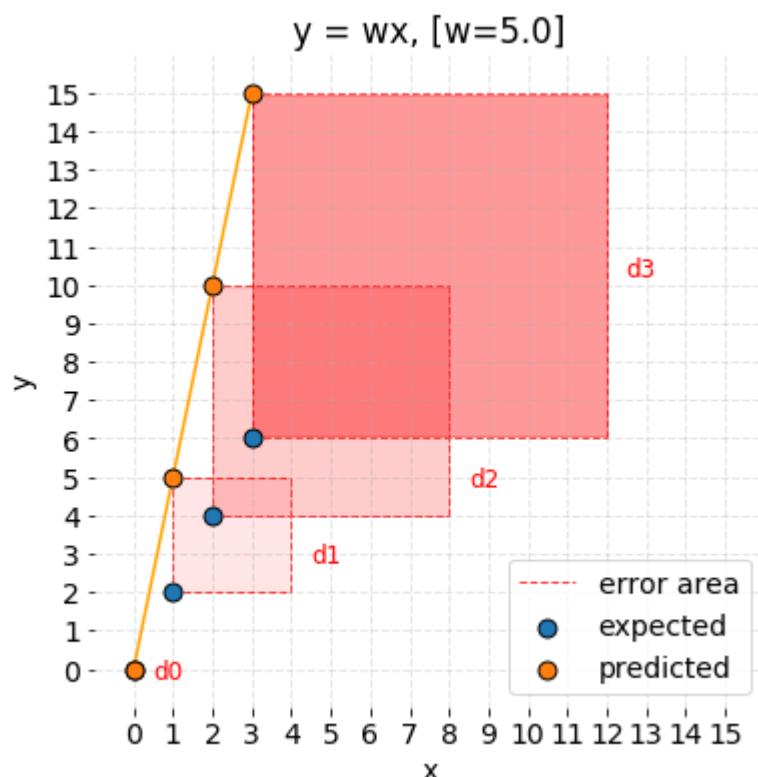
تابع هزینه MSE که مخفف Mean Squared Error می‌باشد، در حقیقت یکی از پرکاربردترین توابع مورد استفاده در مسائل Regression می‌باشد کاری که این تابع انجام می‌دهد این است که می‌آید و میانگین تفاضل مربعات میان داده‌های پیش‌بینی شده و داده‌های واقعی را محاسبه می‌کند. به عبارت دیگر، نوعی دیگر از MAE هست که به جای محاسبه قدرمطلق تفاضل‌ها از مربع آن‌ها استفاده می‌کند.

فرمول آن مطابق زیر است:

$$MSE = \frac{1}{2m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2$$

شکل 2: فرمول مربوط به تابع MSE

در کدام از خطاهای در واقع با فاصله بین نقاط در دستگاه مختصات برابر است. در MSE در نهایت تمامی این فواصل با هم جمع زده می‌شوند و در آخر میان آن‌ها میانگین گرفته می‌شود. در زیر می‌توانید تصویر آن را مشاهده کنید:



شکل 3: نحوه محاسبه MSE

Classification Problems -

:Hinge Loss/Multi class SVM Loss .1

به زبان ساده موردنی که این نوع Loss بیان می‌کند این است که امتیاز دسته‌بندی شده‌های درست باید با یک میزانی Margin به عنوان Safety Margin از مجموع امتیازات همه دسته‌بندی‌های اشتباه بیشتر باشد. به دلیل وجود این فاکتور Margin در این روش، بیشتر از این تابع برای مسائل SVM یا Support Vector Machine استفاده می‌شود. این یک تابع Convex هست و این موضوع کار کردن با آن را در ها آسان می‌کند. فرمول آن را نیز می‌توانید در زیر مشاهده کنید: Convex Optimizer

$$SVM\text{Loss} = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

شکل 4: فرمول روش گفته شده

:Cross Entropy Loss/Negative Log Likelihood .2

این تابع پرکاربردترین تابع در مسائل Classification است. مقدار Cross Entropy زمانی که احتمال پیش‌بینی شده از Label واقعی فاصله می‌گیرد، افزایش می‌یابد. فرمول آن را می‌توانید در زیر مشاهده کنید:

$$CrossEntropyLoss = -(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

شکل 5: فرمول Cross Entropy Loss

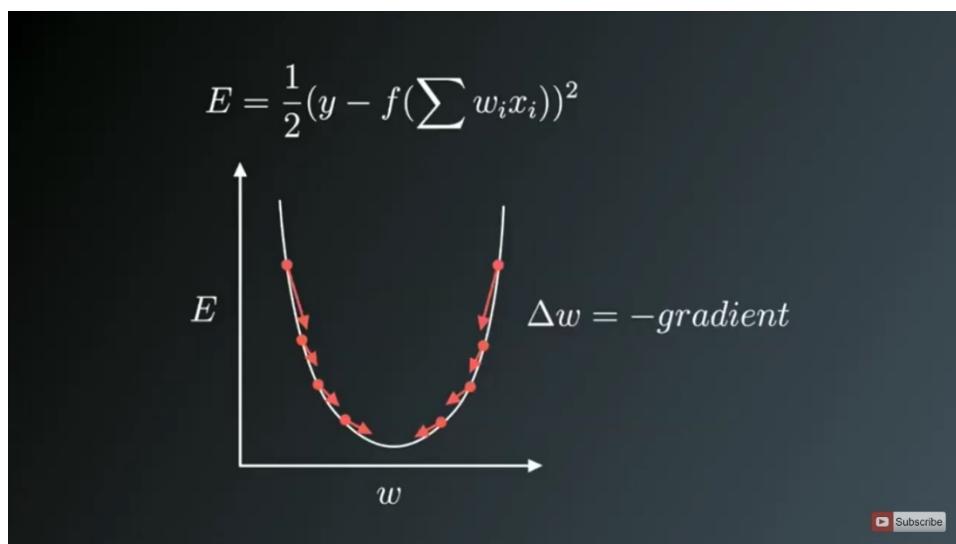
مورد دیگری که در Cross Entropy Loss لازم است که ذکر کنم، این است که این روش به شدت پیش‌بینی‌هایی را که بسیار مطمئن هستند اما اشتباه هستند را جریمه می‌کند.

سوال ۲ - بخش تشریحی

دسته اول توابع بهینه سازی که به نام First Order Optimization Algorithms نیز شناخته می‌شوند، این توابع سعی می‌کنند که یک Loss Function مانند $E(x)$ را با استفاده از مقادیر Gradient پارامترها ماسکیمیم یا مینیمم کنند. پرکاربردترین الگوریتم این دسته Gradient Descent می‌باشد. مشتق مرتبه اول به ما می‌گوید که در یک نقطه تابع ما در حال افزایش یا کاهش هست و اساساً مشتق مرتبه اول به ما یک خط را می‌دهد که به یک نقطه در سطح Error آن مماس می‌باشد.

یک گرادیان نیز یک وکتوری چند مقداره هست که میزان تغییرات y نسبت به x را به ما نشان می‌دهد.

مثال معروف گرادیان، Gradient Descent هست که در زیر می‌توانید شکل عمل کرد آن را مشاهده نمایید و همان‌طور که از اسم آن پیدا هست مرحله به مرحله گرادیان را کاهش می‌دهد تا به جواب برسد.



شکل ۶: نحوه کارکرد Gradient Descent

همچنین Stochastic Gradient Descent و Mini batch Gradient Descent نیز نمونه‌های دیگری از این روش هستند.

روش‌های پیشرفته‌تری نیز مانند Momentum و Adagrad و RMSProb نیز هستند که کمی هوشمندتر عمل می‌کنند اما بیس آن‌ها همین روش کاهش گرادیان هست. و بهترین روش هم که تقریباً در اکثر موارد بهتر عمل می‌کند روش Adam هست.

دسته دوم توابع بهینه سازی که به نام Second Order Optimization Algorithms نیز شناخته می‌شوند، این توابع از مشتق مرتبه دوم استفاده می‌کنند که به نام Hessian نیز شناخته می‌شوند برای این که مقدار تابع Loss را Minimize یا Maximize کنند. Hessian در واقع یک ماتریس از مشتقات جزئی مرتبه دوم می‌باشد. به دلیل این که استفاده از مشتق مرتبه دوم و Hessian بسیار هزینه‌بر است از نظر زمان محاسبه، به این دلیل از آن خیلی استفاده نمی‌شود و ما تاکنون در روش‌هایی که استفاده کردہ‌ایم، آن را نداشته‌ایم. مشتق مرتبه دوم به ما می‌گوید که آیا مشتق مرتبه اول در حال کاهش یا افزایش است و برای ما انحنایها و خمیدگی‌ها را در تابع مشخص می‌کند.

برتری مهم این دسته دوم توابع بهینه‌سازی انحنای‌های سطح را نیز در نظر می‌گیرند و آن‌ها را حذف نمی‌کنند. یکی از مثال‌های این دسته بهینه‌سازی مرتبه دومی نیوتونی یا روش Newton's Second Order Optimization می‌باشد.

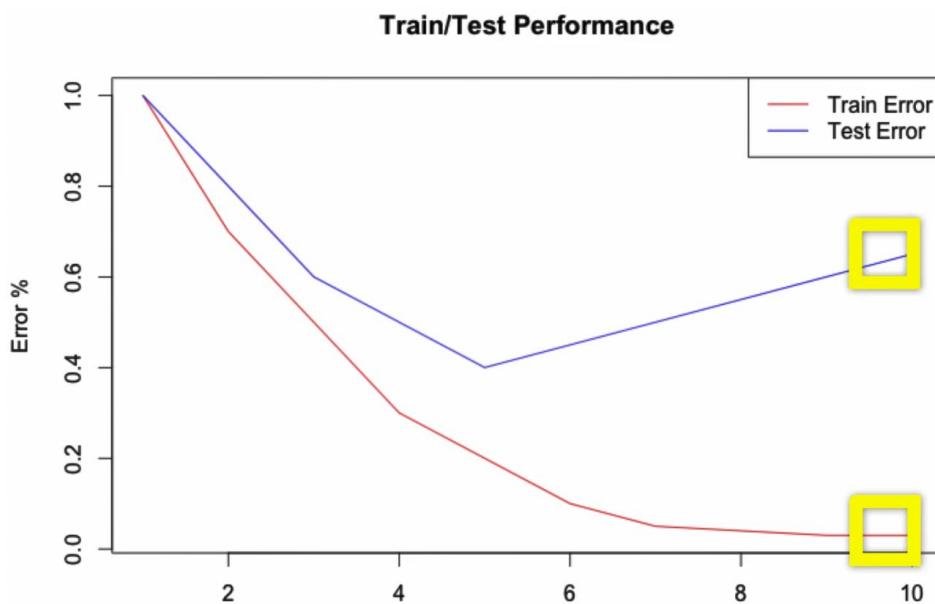
سوال 3 – بخش تشریحی

در این سوال از ما خواسته شده است که در مورد Overfitting توضیحاتی را بدهیم.

به طور ساده بخواهم توضیح بدهم، وقتی روی می‌دهد که داده‌های ما به خوبی، داده‌های Train را یاد بگیرند و بتوانند خطای کمی را در این داده‌ها به ما بدهند اما در داده‌های Validation نتوانند نتایج خوبی را به ما بدهند. به عنوان مثال این حالت مانند دانش‌آموزی می‌ماند که در یادگیری درسش برای امتحان سوالات و مثال‌ها را حفظ می‌کند و آن سوال‌ها را می‌تواند به خوبی جواب بدهد اما در امتحان و آزمون وقتی با سوالات جدید مواجه می‌شود دچار مشکل می‌شود.

به نوعی ما می‌توانیم بگوییم که در Overfitting شبکه ما به سمت حفظ کردن داده‌های آموزشی می‌رود و توانایی Generalization خود را برای سایر داده‌ها از دست می‌دهد.

همچنین وقتی ما پیچیدگی و تعداد لایه‌های یک شبکه عصبی عمیق را نسبت به سختی داده‌هایمان، زیاد می‌کنیم احتمال این که شبکه ما به سمت حفظ کردن جزئیات داده‌های آموزشی برود را زیاد کردہ‌ایم و در نتیجه احتمال این که Overfitting روی بدهد، بیشتر می‌شود. در زیر می‌توانید نمونه آن را ببینید:



شکل 7: نمونه‌ای از Overfitting

راه‌های رفع Overfitting

استفاده از epoch‌های کمتر، یا موردنامه که ما آن را به عنوان early stopping نیز می‌شناسیم که در آن همان‌طور که در شکل بالا نیز می‌توانید مشاهده کنید ما قبل از این که شبکه ما به سمت حفظ ویژگی‌ها برود و مقدار خطای آن مجدداً شروع به افزایش کند، فرآیند آموزش را متوقف می‌کنیم و به این شکل می‌توانیم که جلوی overfit شدن آن را بگیریم.

2- استفاده از regularization: برای حل مشکل overfitting می‌توانیم که regularization وزن‌ها را به مدل‌مان بدهیم. این کار باعث می‌شود که برای وزن‌های زیاد، یک هزینه‌ای را به تابع loss مامان اضافه کنیم. با انجام این کار، ما یک مدل ساده‌تری را دریافت می‌کنیم که مجبور است که تنها الگوهای مرتبط را در داده‌های آموزشی را یاد بگیرد.

کاری که ما در Regularization انجام می‌دهیم این است که می‌آییم و با دو روش L1 و L2 سعی می‌کنیم که مقدار این وزن‌ها را کنترل کنیم و نگذاریم که مقدار وزن‌ها خیلی زیاد شود که کنترل آن‌ها از دست ما خارج شود.

دو نوع regularization به نام‌های L2 regularization و L1 regularization وجود دارند.

L1 Regularization

$$R(W) = \sum_k \sum_l |W_{k,l}|$$

L1-Regularization

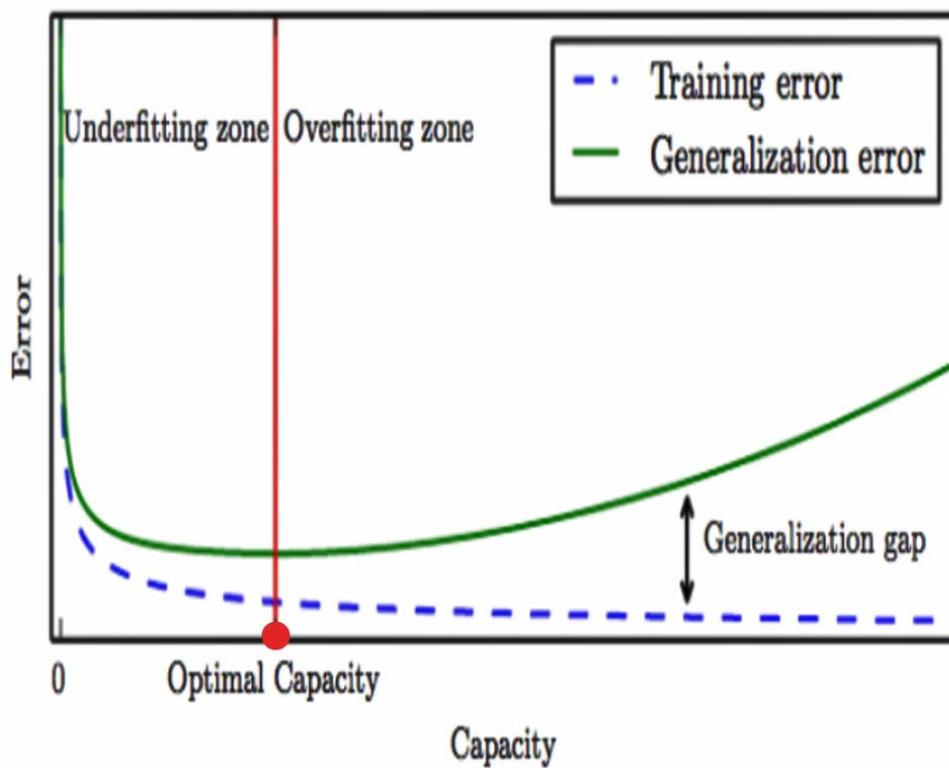
شکل 8: نحوه انجام L1 Regularization بر روی وزن‌ها

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

L2-Regularization

شکل 9: نحوه انجام L2 Regularization بر روی وزن‌ها

معمولاً از L2 استفاده می‌شود زیرا ترجیح ما این است که تابعمان مشتق پذیر باشد و L2 مشتق پذیر است.



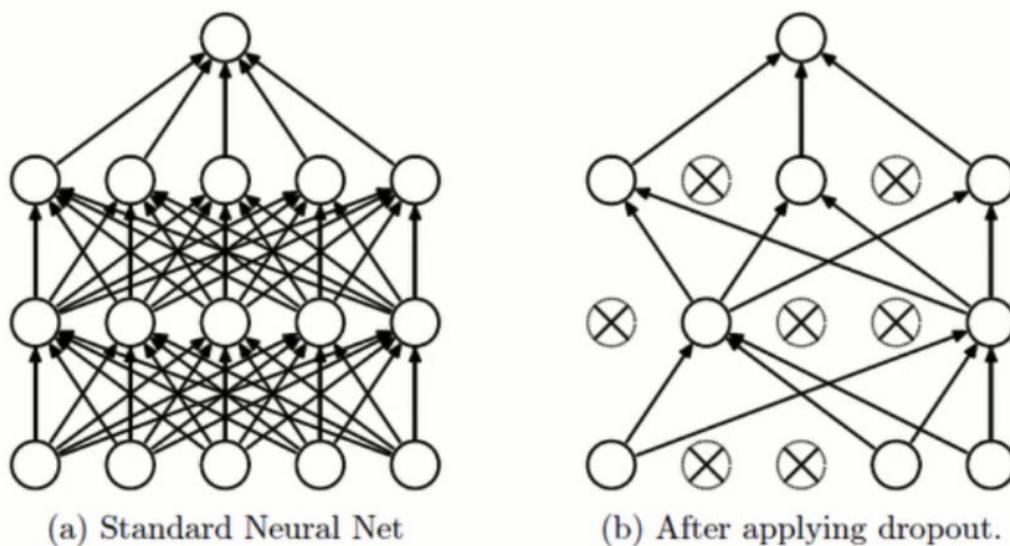
شکل 10: مناسب برای overfit نشدن شبکه Capacity

3- استفاده از دیتاست های بزرگ تر:

علاوه بر موارد گفته شده در بالا در بعضی از اوقات، استفاده از یک دیتاست بزرگ تر و با داده های بیشتر می تواند که باعث شود شبکه ما به سمت حفظ کردن نرود و overfit نشود. اما این روش همواره جواب نمی دهد و در بعضی از مواقع اگر جواب بدید باعث افزایش دقت ما نیز می شود البته تا زمانی که دیتاهای اضافه شده noisy نباشند و clean و relevant باشند.

4- استفاده از DropOut:

علاوه بر موارد گفته شده در بالا خیلی از موقع استفاده از Dropout نیز می تواند در کم کردن overfit موثر باشد و احتمال آن را کاهش دهد و نحوه عملکرد dropout به این صورت است که با یک احتمالی که به این تابع می دهیم، می آید و تعدادی از نودها را به صورت تصادفی در هر مرحله از train حذف می کند و با این کار باعث می شود که در هر مرحله کمی از ویژگی ها را حذف کنیم و به این صورت از حالتی که شبکه شروع به حفظ کردن کند، جلوگیری می کنیم و اجازه می دهیم که وزن ها بهتر در طول شبکه پخش شوند.



شکل 11: نحوه کارکرد Dropout

معمولًا استفاده از Dropout در لایه های آخر شبکه مناسب است.

همچنین لازم است اشاره کنم که این کار موقع Train انجام می شود و موقع Test دیگر انجام نمی شود. چون موقع train روی می دهد و در این موقع است که کمک می کند.

معمولًا هم احتمال Dropout برابر با 0.5 بهترین عملکرد را دارد طبق مقاله ای که روی این زمینه کار کرده است.

5- کم کردن عمق و پیچیدگی شبکه: با انجام این کار، باعث می‌شویم که شبکه ما به سمت حفظ کردن داده‌ها نرود.

6- کم کردن تعداد node‌ها در هر لایه: با انجام این کار نیز می‌توانیم خیلی از اوقات جلوی overfit شدن را بگیریم و باعث شویم که شبکه ما به سمت حفظ کردن ویژگی‌ها نرود.

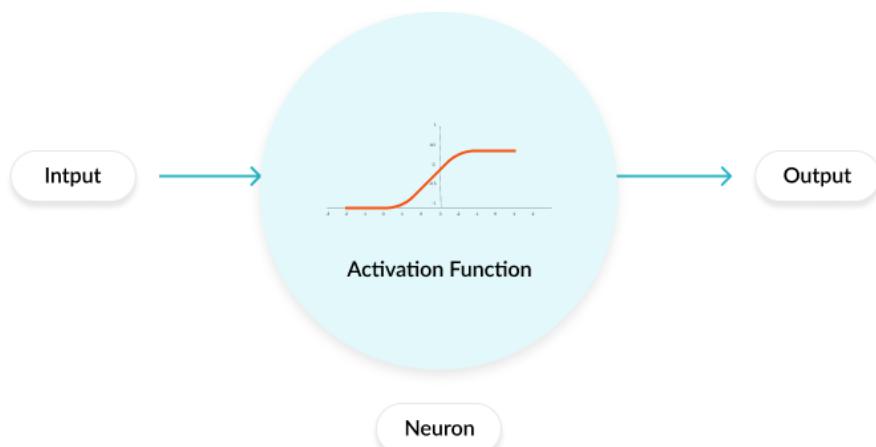
7- استفاده از Data Augmentation: همان‌طور که پیش‌تر گفتیم استفاده از یک دیتاست بزرگ‌تر مناسب می‌تواند راه خوبی برای کاهش overfitting باشد و یک راه مناسب دیگر بیشتر کردن تعداد دیتاهای همان دیتاست موجود با استفاده از Data Augmentation هست که می‌تواند در خیلی از موارد باعث کم شدن Overfit می‌شود. (در سوال ۵ کامل در مورد نحوه عملکرد این روش توضیح داده‌ام)

سوال 4 – بخش تشریحی

در این قسمت از ما خواسته شده است که مشکلات استفاده از Activation Function خطی را بیان کنیم و سپس جایگزین‌هایی را برای آن ارائه دهیم.

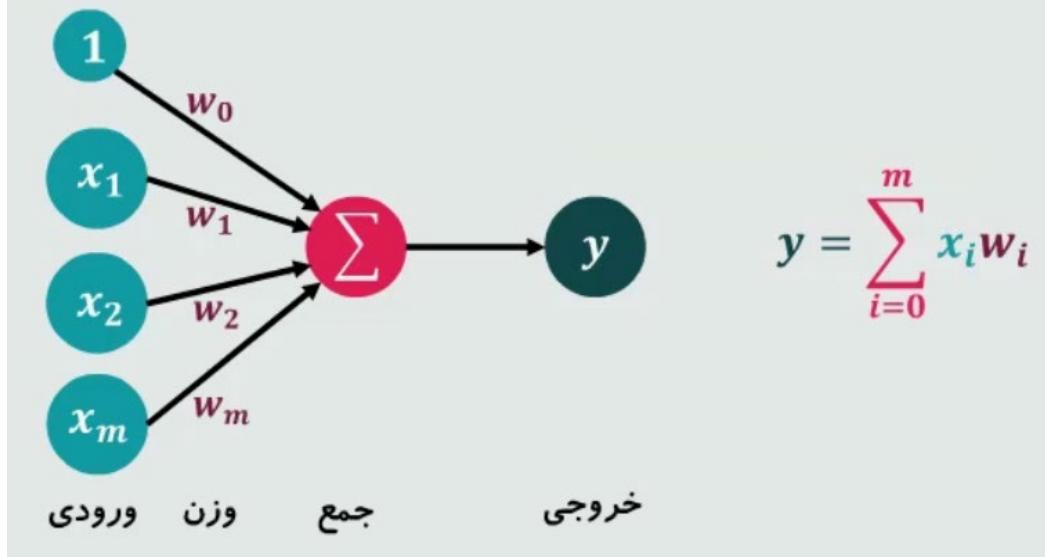
ابتدا من سعی می‌کنم تعاریفی از علت این که ما اصلاً از Activation Function استفاده می‌کنیم را ارائه دهم و سپس به بیان مشکلات موجود در Activation Function‌های خطی بپردازم و سپس جایگزین‌های آن را معرفی کنم.

Activation Function در واقع به هر نرون در شبکه اضافه می‌شود تا این که مشخص کند که آیا باید این نرون مورد نظر به اصطلاح fire یا اجرا بشود یا نه نباید فعال باشد و این کار را بر اساس این که آیا ورودی هر نرون مرتبط با پیش‌بینی مدل ما هست انجام می‌دهد. همچنین Activation Function‌ها همچنین به ما کمک می‌کنند تا خروجی هر نرون را نرمال‌سازی هم بکنیم. همچنین اثراتی را در زمینه بار محاسبات کامپیوتی هم دارند که من دیگر به آن نمی‌پردازم.



شکل 12: نحوه عملکرد Activation Function ها

حال می خواهم به بیان مشکل استفاده از Activation Function بپردازم. و برای این کار اول حالتی که نداریم را بررسی می کنم:
ما Activation Function در حالت عادی و هنگامی که نباید، نباشد، ما نرون هایمان به شکل زیر هستند:



شکل 13: عملکرد شبکه بدون حضور Activation Function

و در این حالت نرون ما یک خاصیت خطی ساده دارد و نمی توانیم آن را انعطاف پذیر کنیم و خاصیت به آن بدھیم و صرفا یک تابع عادی داریم مانند شکل زیر برای یک شبکه عصبی سه لایه:

$$s = W_3(W_2(W_1x))$$

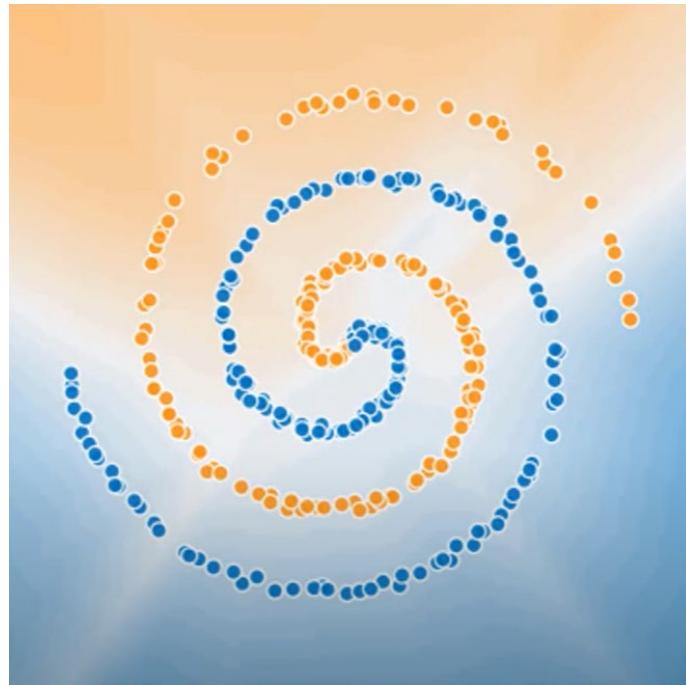
شکل 14: حالت بدون استفاده از Activation Function در شبکه عصبی سه لایه

و سپس می‌توانیم به سادگی نشان دهیم که این صرفاً یک تابع معمولی هست:

$$\begin{aligned} s &= W_3(W_2(W_1x)) \\ &= (W_3W_2W_1)x \\ &= Wx \end{aligned}$$

شکل 15

و همان‌طور که در شکل بالا می‌توانید مشاهده کنید، این تابع صرفاً معادل یک نرون ساده برای ما می‌شود و نمی‌تواند خواصی را برای ما اضافه کند تا در محیط‌های پیچیده مانند شکل زیر تصمیم‌گیری بکنیم.



شکل 16: یک تقسیم بند نسبتاً پیچیده که در صورت نبود Activation Function ما نمی‌توانیم به خوبی دسته‌بندی کنیم

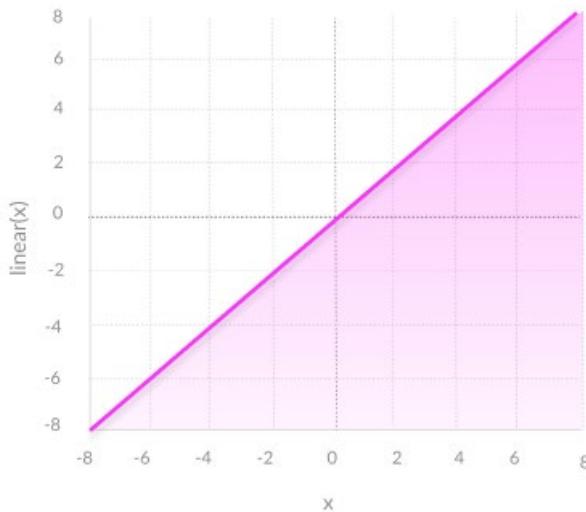
حال ما می‌آییم به همان معادله‌ای که در بالا داشتیم، Activation Function را اضافه می‌کنیم و اکنون معادله آن به شکل زیر می‌شود:

$$s = W_3 f(W_2 f(W_1 x))$$

شکل 17: اضافه شدن Activation Function به یک شبکه عصبی سه لایه

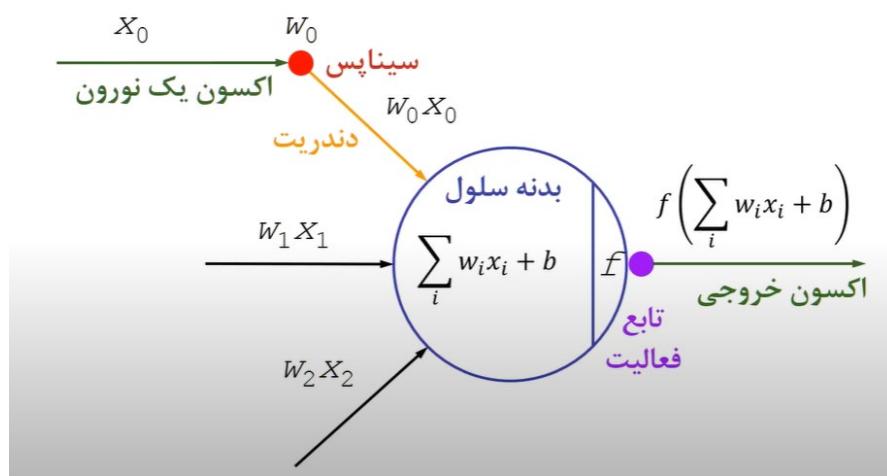
حال اگر در این حالت Activation Function ما خطی باشد یعنی مانند شکل زیر:

$$\mathbf{A} = \mathbf{c}\mathbf{x}$$



شکل 18: Activation Function خطی ساده

همان طور که در شکل بالا می بینید این Activation Function عملکرد آن مانند تابع همانی همان $y=f(x)$ می باشد و در واقع در معادله و ورودی هایش هیچ تغییری را ایجاد نمی کند و همان ورودی را به عنوان خروجی به ما می دهد و عملاً تغییر خاصی را برای ما ایجاد نمی کند که مدل را یادگیری ویژگی های پیچیده مناسب کند و عملاً شبکه عصبی ما را مجدد به یک دسته بند خطی ساده تبدیل می کند و هیچ ویژگی غیر خطی به آن نمی دهد و انگار ما اصلاً تابع f ای مانند شکل زیر نداریم:



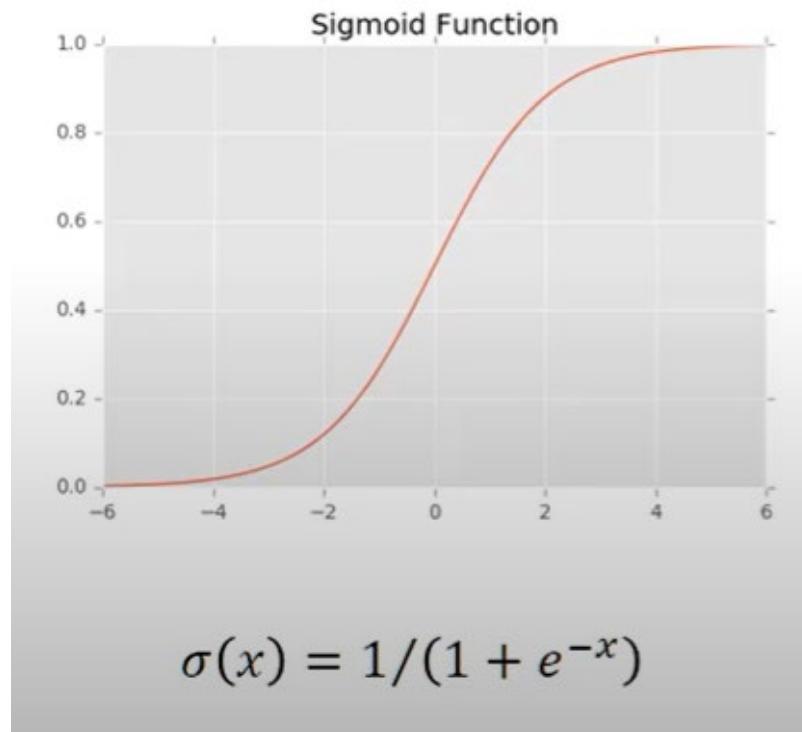
شکل 19: اعمال Activation Function بر روی نرون ها

خب پس بنابراین استفاده از Activation Function خطی برای ما عملاً هیچ فایده های ندارد و تغییری ایجاد نمی کند و عملاً تبدیل به یک Classifier خطی برای ما می شود که برای مسائل خطی کار می کند

اما مسائل دنیای واقعی پیچیدگی بسیار بیشتر از این را دارند و این تابع اصلاً کفايت نمی‌کند و پاسخگو نیست و باید به سراغ جایگزین‌هایی برای آن برویم که در ادامه به تعدادی از آن‌ها اشاره می‌کنم:

1. تابع سیگموید (Sigmoid Function)

تابع سیگموید که در شکل زیر فرمول آن را به همراه نمایش مشاهده می‌کنید جزو اولین Activation هایی بود که به کار گرفته شد اما اکنون دیگر خیلی از آن استفاده نمی‌شود.



شکل 20: نمایش و فرمول تابع سیگموید

مزایا:

- این تابع همواره مشتق‌پذیر است.
- از روی دادن پرش‌ها و jump در مقادیر خروجی جلوگیری می‌کند.
- باعث می‌شود که در هر نرون خروجی‌های آن بین 0 و 1 نرمال شوند.
- به این معنا که برای مقادیر بیشتر از 2 و کمتر از -2 سعی می‌کند که مقادیر y را به نزدیکی 1 و 0 برساند و این باعث شفاف شدن پیش‌بینی‌ها می‌شود و ما را با احتمالات و... کمتر درگیر می‌کند.

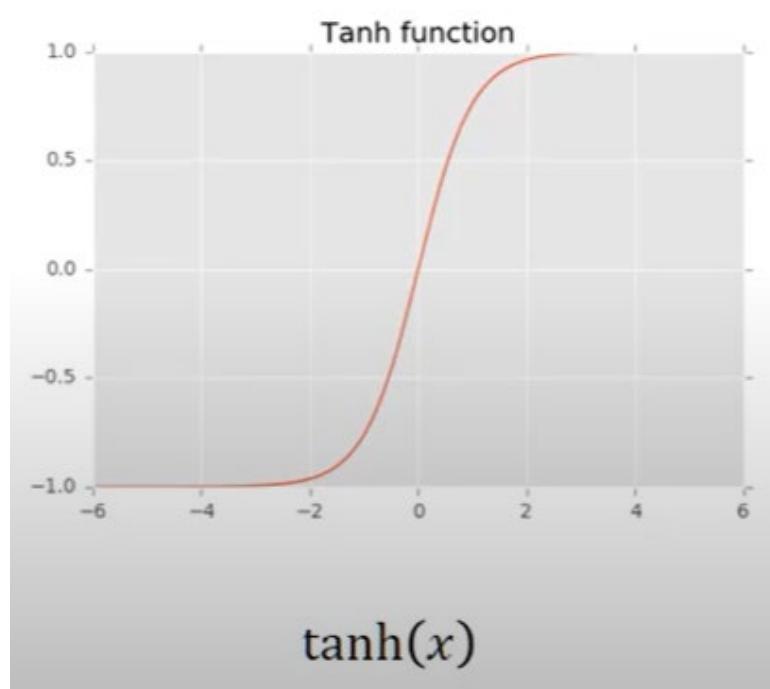
معایب:

- بزرگترین و اصلی ترین مشکل این تابع هست و به این معنا که برای Vanishing Gradient مقادیر خیلی بزرگ و خیلی کوچک عملای هیچ تغییری در پیش‌بینی که انجام می‌دهد وجود ندارد و عملای آموزشی نمی‌بیند چون که مشتق (شیب) در این محل‌ها عملای خیلی نزدیک به صفر هست. (نرون‌های اشباع شده، گرادیان را از بین می‌برد.)
- این موضوع که خروجی‌های تابع سیگموید دارای میانگین صفر نیست و به اصطلاح Zero Centered نیست و خروجی آن bipolar نیست.
- هزینه‌بر بودن از نظر محاسبه زیرا محاسبه این تابع کمی سنگین است و زمان‌بر می‌شود.

2- تابع (Hyperbolic Tangent Function) Tanh

این تابع که شکل آن را می‌توانید در زیر مشاهده کنید، بسیار شکل کلی آن شبیه به سیگموید می‌باشد:

کاربرد این تابع بیشتر در پردازش متن است.



شکل 21: نمایش تابع Tanh

همان طور که در شکل بالا مشخص است، شکل کلی این تابع شبیه به سیگموید است با این تفاوت که بین ۱ و -۱ است به جای ۰ و ۱.

مزایا:

- همان مزایای سیگموید + Zero Centered هست و مرکز آن صفر هست.

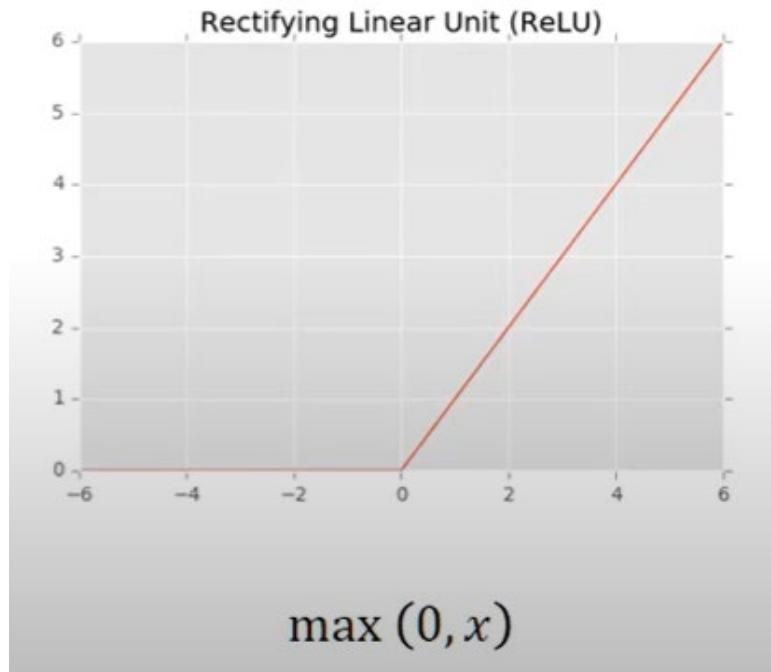
معایب:

- همان معایب سیگموید منهای آن مشکلی که مرکز آن صفر نیست. یعنی همچنان مشکل اشباع شدن را دارد و در مقادرهای خیلی بزرگ یا خیلی کوچک عملای شبکه ما به دلیل شبیه صفر همان طور که در شکل مشاهده می‌کنید. نمی‌تواند که یاد بگیرد.

: (Rectified Linear Unit) ReLU

در شکل زیر می‌توانید شمای آن را به همراه معادله‌اش مشاهده نمایید:
عملکرد این تابع بسیار ساده است و در قسمت منفی‌ها مقدار آن صفر و در قسمت مثبت‌ها دقیقاً مانند تابع همانی عمل می‌کند.

ReLU



شکل 22: نمایش تابع ReLU

مزایا:

- Computational Efficiency و کارایی خوب از نظر محاسباتی و که این دلیل که همان‌طور که از معادله و شکل آن مشخص است، عملکرد بسیار ساده‌ای دارد و در قسمت‌های منفی تماماً صفر و در مثبت‌ها دقیقاً خود مقدار را برمی‌گردد.
- غیر خطی بودن: اگر چه شبیه به توابع خطی هست، اما در واقع غیر خطی هست و بنابراین اجازه می‌هد به ما که مشتق بگیریم از آن و برای عمل backpropagation از آن استفاده کنیم.
- در نواحی مثبت اشباع نمی‌شود و گرادیان آن زیاد می‌شود که این موضوع باعث می‌شود که آموزش ببیند.

معایب:

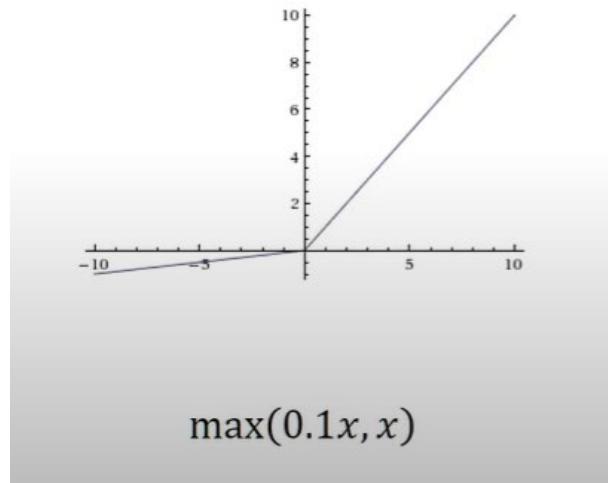
- مشکل Dying ReLU: هنگامی که ورودی‌ها به صفر نزدیک می‌شوند یا منفی می‌شوند، گرادیان تابع صفر می‌شود و بنابراین در این قسمت‌های عمل backpropagation نمی‌تواند انجام شود و شبکه در این قسمت‌ها یاد نمی‌گیرد.

- همچنان این تابع نیز Sigmoid نیست اما به میزان کمتری نسبت به Sigmoid
- مشکل نداشتن مشتق در صفر که البته خیلی مشکل بزرگی نیست اولاً چون خیلی کم پیش می‌آید و ثانیاً چون می‌توانیم این مشکل را با Sub Gradient حل کنیم.

در آزمایش‌های مختلف نشان داده شده است که با همین یک تغییر ساده م استفاده از ReLU به جای سایر توابع، شبکه ما ۵ تا ۶ برابر سریع‌تر آموزش می‌بیند.

توابع دیگری نیز هستند همانند Leaky ReLU که در زیر شکل آن را می‌بینید اما من به جزئیات آن‌ها اشاره نمی‌کنم. معمولاً از تابع ReLU ما استفاده می‌کنیم و این تابع عملکرد بسیار خوبی را از خود نشان می‌دهد.

Leaky ReLU

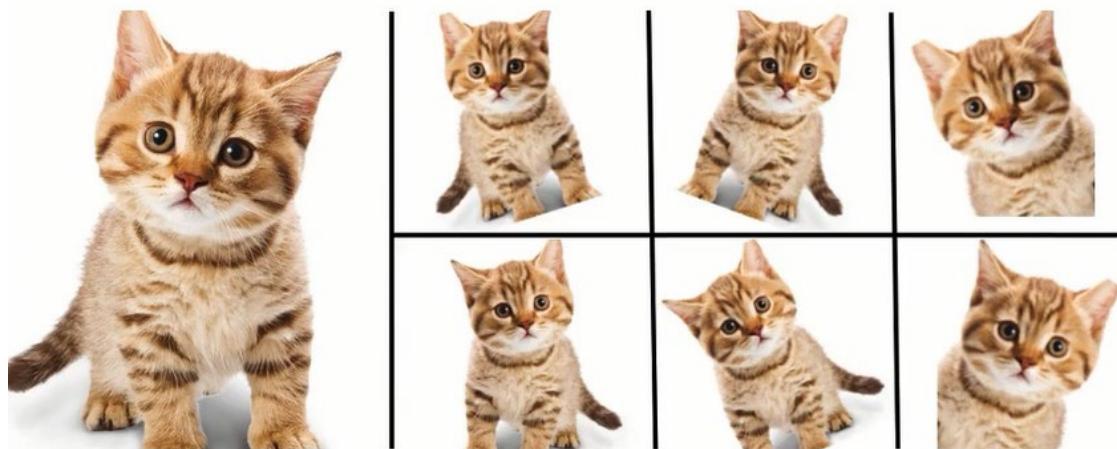


شکل 23: تابع Leaky ReLU

سوال 5 – بخش تشریحی

در این سوال از ما خواسته شده است که در مورد مفهوم Data Augmentation توضیحاتی را ارائه دهیم.
به صورت کلی Data Augmentation به فرآیندی می‌گویند که ما در آن دیتاهای جدیدتری را برای مدلمان طراحی می‌کنیم تا در طول پروسه training از آن استفاده کنیم.

این کار را با استفاده از دیتاست موجودمان انجام می‌دهیم که آن را گرفته و آن‌ها را در جهت استفاده transform می‌کنیم و تغییر می‌دهیم تا تصاویر جدیدی را برای آموزش شبکه‌مان ایجاد کنیم. یک سری از نمونه‌های انجام آن را بر روی یک تصویر گربه می‌توانید در شکل زیر مشاهده نمایید:

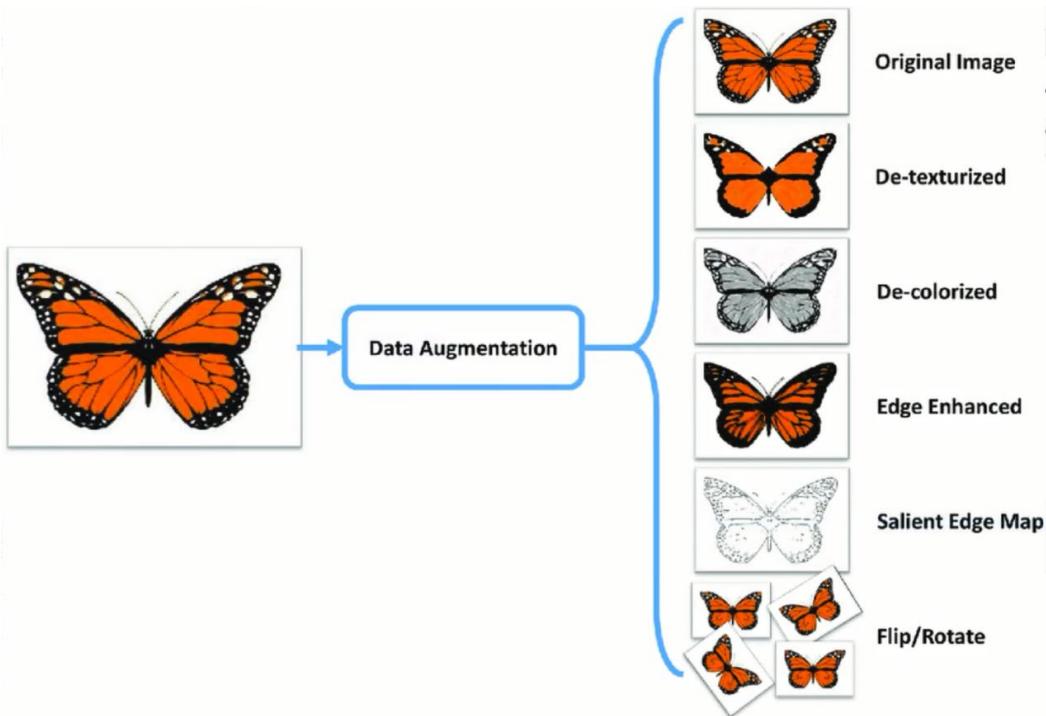


Enlarge your Dataset

شکل 24: اعمال Data Augmentation بر روی تصویر یک گربه

نمونه‌هایی از transform کردن تصاویر می‌تواند rotate کردن تصویر یا zoom کردن در تصویر یا هردو با هم باشد.

این تصاویر جدید که با کارهایی که گفته شد ایجاد می‌شوند را به عنوان Augmented Image ها می‌شناسیم و به ما اجازه می‌دهند که با اضافه کردن این موارد به دیتاستمان آن را augment کنیم.



شکل 25: پروسه انجام Data Augmentation

استفاده از Data Augmentation در یک دیتاست باعث می‌شود، مدلی از شبکه عصبی که طراحی کرده‌ایم به هر عکس از منظرها و Perspective‌های مختلفی نگاه کند و این موضوع به آن اجزه می‌دهد تا ویژگی‌های مناسب را با دقت بهتری استخراج کند و همچنین ویژگی‌های بیشتری از هر داده به دست بیاورد. استفاده از این روش باعث می‌شود که شبکه طراحی شده را نیز Boost کنیم و بهتر کنیم.

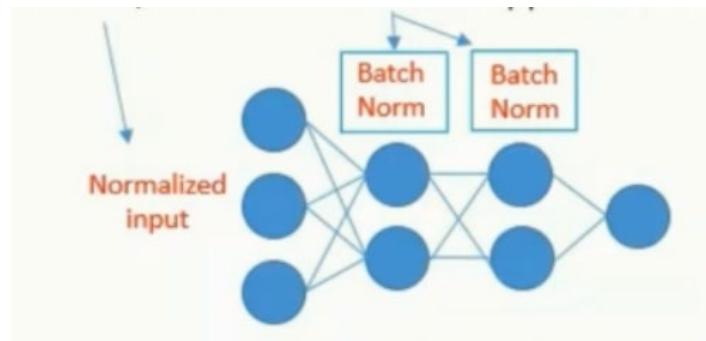
سوال 6 – بخش تشریحی

در این قسمت از سوال از ما خواسته شده است که در مورد Batch Normalization توضیح دهیم.

ما میدانیم که ورودی‌های شبکه عصبی را برای جلوگیری از خطا رفتن و دقت بهتر خوب است که نرمال کنیم حال اگر سایر لایه‌های مخفی بعضی از نرون‌ها مقادیر زیادی پیدا کنند، باز مجدد باعث می‌شود که یادگیری شبکه عصبی ما un stable شود و به مشکل بر می‌خوریم. بنابراین منطق اصلی کاری که در Batch Normalization انجام می‌دهیم این است که ما ورودی‌ها را در هر لایه نرمال‌سازی می‌کنیم و ما این نرمال‌سازی را مطابق با داده‌های ورودی از هر Batch از داده‌ها که معین کرده‌ایم انجام می‌دهیم به همین دلیل به آن Batch Normalization می‌گویند!

بنابراین در این روش ما هر نرون در هر لایه را مطابق با هر Batch از نمونه‌ها نرمال‌سازی می‌کنیم. میانگین و انحراف معیار برای هر Batch از داده‌ها به این منظور محاسبه می‌شوند.

پس می‌شود نرمال‌سازی نرون خروجی در هر لایه مخفی به جای نرمال‌سازی فقط برای لایه ورودی برای هر Batch از داده. به بیان دیگر این روش مقادیر لایه مخفی را که خیلی پراکنده شده‌اند را Shift می‌دهد.



شکل 26: اعمال Batch Normalization بر روی لایه‌های مخفی

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;
 Parameters to be learned: γ, β
Output: $\{y_i = BN_{\gamma, \beta}(x_i)\}$

$$\begin{aligned}\mu_{\mathcal{B}} &\leftarrow \frac{1}{m} \sum_{i=1}^m x_i && // \text{mini-batch mean} \\ \sigma_{\mathcal{B}}^2 &\leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 && // \text{mini-batch variance} \\ \hat{x}_i &\leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} && // \text{normalize} \\ y_i &\leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i) && // \text{scale and shift}\end{aligned}$$

شکل 27: ترتیب مراحل اعمال Batch Normalization بر روی داده‌ها

در شکل بالا می‌توانید ترتیب و نحوه اعمال و محاسبه Normalization برای هر Batch از داده‌ها را در هر لایه مخفی مشاهده کنید. علت قرار دادن مقدار اپسیلون در کسر محاسبه نرمال‌سازی در شکل بالا نیز جلوگیری از تقسیم شدن این عبارت بر صفر هست. مقادیر گاما و بتا هم در فرمول بالا که مشخص شده‌اند،

Learnable Parameters نام دارند و این مقادیر را در خودش شبکه یاد می‌گیرد تا ببیند کدام ضریب برای اعمال نرمال سازی مناسب‌تر است.

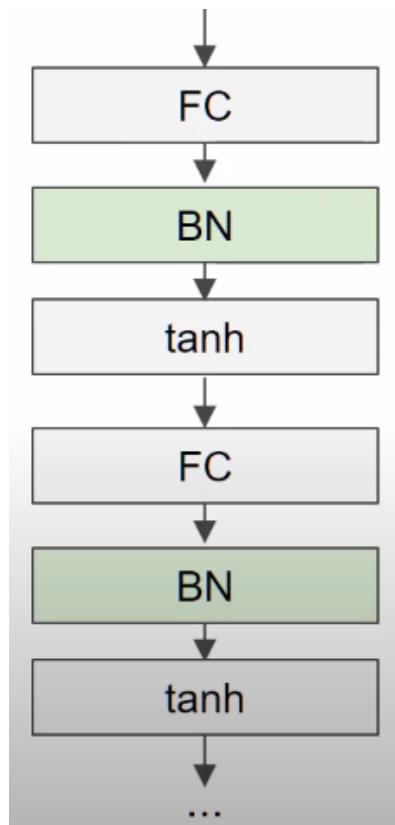
در واقع هر بعد را به یک توزیع گاووسی نرمال تبدیل می‌کنیم با میانگین 0 و انحراف معیار 1.

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

شکل 28: فرمول محاسبه Batch Normalization

ما برای محاسبه باید بیاییم و میانگین و واریانس را برای هر بعد به طور مستقل محاسبه کنیم و سپس در آخر نرمال سازی را انجام دهیم. و ما این محاسبات را به ازای هر Batch انجام می‌دهیم به این صورت که اگر 100 اندازه Batch ما باشد، سپس این 100 تا داده هر بار به شبکه داده می‌شود و روی این 100 تا انجام می‌دهیم.

اعمال Batch Normalization را باید قبل از اعمال Activation function انجام دهیم. مانند آن را در شکل زیر می‌توانید مشاهده کنید:



شکل 29: اعمال Batch Normalization

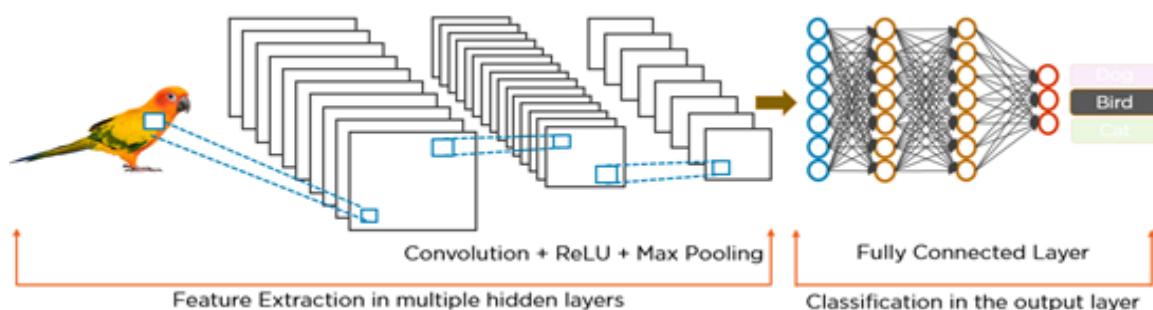
اعمال Batch Normalization معمولاً پس از لایه‌های Fully Connected و پیش از تابع فعالیت غیر خطی انجام می‌شود.

با عث می‌شود که جریان گرادیان در شبکه بهبود پیدا کند. همچنین امکان استفاده از مقادیر بزرگ را برای نرخ یادگیری برای ما فراهم می‌کند. وابستگی به مقداردهی اولیه وزن‌ها را کاهش می‌دهد. به عنوان نوعی تنظیم کننده نیز در شبکه عمل می‌کند و در نتیجه نیاز به استفاده از Dropout که پیشتر توضیح داده شد را کاهش می‌دهد.

بخش عملی

در این بخش از ما خواسته شده است که عمل دسته‌بندی داده‌های دیتابست معروف و نسبتاً پیچیده German Terrific Signs را انجام بدیم که دیتابستی مت Shank از 43 تا دسته تصویر که مربوط به علائم رانندگی هست. این دستاست نسبت به سایر دیتابست‌هایی که تاکنون بررسی کردہ‌ایم کمی پیچیده‌تر است

و همچنین در این دستاست Distortionها و Disturbanceها نسبتاً زیاد است و تصاویر نویز و پیش‌زمینه و... متفاوتی دارند و درنتیجه به همین دلیل استفاده از روش‌های معمول دسته‌بندی خطی نمی‌تواند به ما خیلی کمک کند و لایه‌های کانولوشنی می‌تواند بسیار در این مورد موثر باشد زیرا که اقدام به اعمال فیلترهای مختلف بر روی تصاویر می‌کند و به این شکل می‌تواند ویژگی‌های مختلف را از تصاویر مانند Edge، Cornerها و Curvatureها را استخراج کند و سپس با استفاده از این موارد Sub-Skeletonها را تشخیص دهد و سپس بتواند یک تصویری از کل شی را در در نظر بگیرد و در آخر کلاس و دسته آن را مشخص کند. این موارد را می‌توانید در شکل زیر برای تشخیص یک کلاس پرنسه مشاهده کنید:



شکل ۳۰: نحوه عملکرد شبکه‌های کانولوشن

ابتداً کار اقدام به لود کردن تصاویر و نمایش آن‌ها می‌کنم:

قبل از توضیح در این مورد لازم است ذکر کنم که برای کار کردن با Google Colab و این که بتوانم به راحتی هر بار به دیتاست دسترسی داشته باشم، اقدام به آپلود دیتاست در ریپازیتوری خودم در گیت کردم که با مراجعه به این [لینک^۱](#) می‌توانید آن را مشاهده نمایید.

و پس از انجام این کار در ابتداً کار در Google Colab به راحتی و با دستور زیر اقدام به دریافت دیتاست می‌کرم و سپس به راحتی می‌توانستم از آن استفاده کنم.

[https://github.com/parhamzm/German-Traffic-Signs-Dataset-GTSRB^۱](https://github.com/parhamzm/German-Traffic-Signs-Dataset-GTSRB)

```
1 !git clone https://github.com/parhamzm/German-Traffic-Signs-Dataset-GTSRB
```

```
Cloning into 'German-Traffic-Signs-Dataset-GTSRB'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 51937 (delta 0), reused 2 (delta 0), pack-reused 51934
Receiving objects: 100% (51937/51937), 299.45 MiB | 34.36 MiB/s, done.
Resolving deltas: 100% (1/1), done.
Checking out files: 100% (51891/51891), done.
```

شکل 31: دریافت دیتاست از Repository گیت خودم در گوگل Colab

لازم به ذکر است که من یک فایل دیگر به نام signnames.csv را نیز به دیتاست خودم اضافه کردم که شامل نام تمامی 43 دسته موجود در این دیتاست است. که می‌توانید در ادامه آن‌ها را مشاهده کنید.

جدول 1: لیست دسته‌های موجود در دیتاست

| ClassId | SignName |
|---------|--|
| 0 | Speed limit (20km/h) |
| 1 | Speed limit (30km/h) |
| 2 | Speed limit (50km/h) |
| 3 | Speed limit (60km/h) |
| 4 | Speed limit (70km/h) |
| 5 | Speed limit (80km/h) |
| 6 | End of speed limit (80km/h) |
| 7 | Speed limit (100km/h) |
| 8 | Speed limit (120km/h) |
| 9 | No passing |
| 10 | No passing for vechiles over 3.5 metric tons |
| 11 | Right-of-way at the next intersection |
| 12 | Priority road |
| 13 | Yield |
| 14 | Stop |
| 15 | No vechiles |
| 16 | Vechiles over 3.5 metric tons prohibited |
| 17 | No entry |
| 18 | General caution |
| 19 | Dangerous curve to the left |
| 20 | Dangerous curve to the right |
| 21 | Double curve |
| 22 | Bumpy road |
| 23 | Slippery road |
| 24 | Road narrows on the right |

| | |
|----|--|
| 25 | Road work |
| 26 | Traffic signals |
| 27 | Pedestrians |
| 28 | Children crossing |
| 29 | Bicycles crossing |
| 30 | Beware of ice/snow |
| 31 | Wild animals crossing |
| 32 | End of all speed and passing limits |
| 33 | Turn right ahead |
| 34 | Turn left ahead |
| 35 | Ahead only |
| 36 | Go straight or right |
| 37 | Go straight or left |
| 38 | Keep right |
| 39 | Keep left |
| 40 | Roundabout mandatory |
| 41 | End of no passing |
| 42 | End of no passing by vehicles over 3.5 metric tons |

پس از این که دیتاست را دریافت کردیم اقدام به آماده‌سازی آن برای استفاده در شبکه عصبی می‌کنم.

در ابتدا با استفاده از دستورات زیر اقدام به نمایش 5 عدد از هر کدام از این کلاس‌ها می‌کنم:

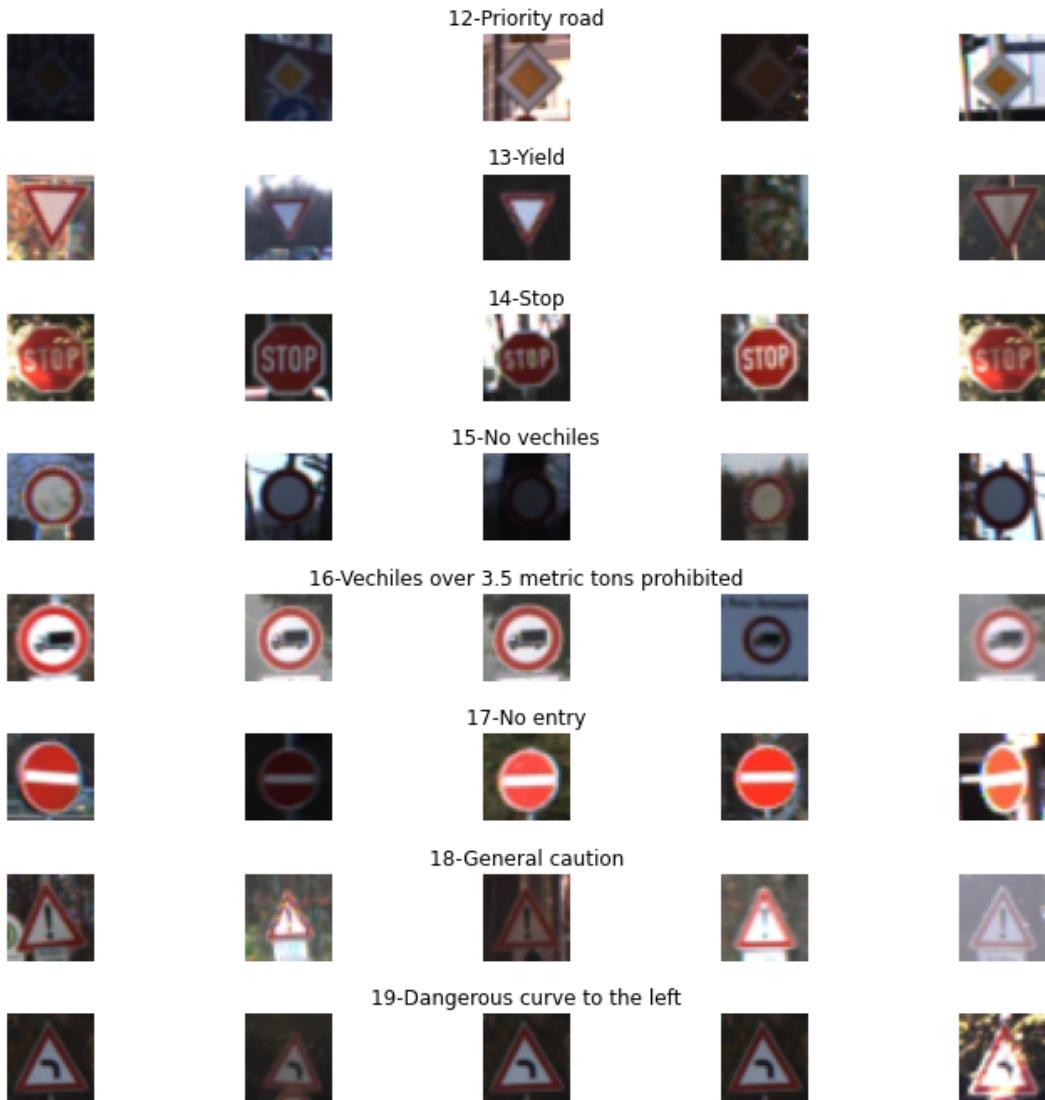
```

1 def img_convert(tensor):
2     image = tensor.clone().detach().numpy()
3     image = image.transpose(1, 2, 0) # we reverse the pixels
4     image = image * np.array([0.5, 0.5, 0.5]) + np.array([0.5, 0.5, 0.5]) # denormalization
5     image = image.clip(0, 1)
6     return image
7
8 data_names = pd.read_csv('German-Traffic-Signs-Dataset-GTSRB/signnames.csv')
9 data = pd.read_csv('German-Traffic-Signs-Dataset-GTSRB/Train.csv')
10
11 num_of_samples = []
12
13 cols = 5
14 num_of_classes = 43
15
16 fig, axs = plt.subplots(nrows=num_of_classes, ncols=cols, figsize=(10, 50))
17 fig.tight_layout()
18
19
20 for i in range(cols):
21     for j, row in data_names.iterrows():
22         x_selected = X_train[y_train == j]
23         x_selected = x_selected.squeeze()
24         axs[j][i].imshow(img_convert(x_selected[random.randint(0, (len(x_selected) - 1)), :, :]), cmap=plt.get_cmap('gray'))
25         axs[j][i].axis("off")
26         if i == 2:
27             axs[j][i].set_title(str(j) + " - " + row["SignName"])
28             num_of_samples.append(len(x_selected))

```

شکل 32: نمایش 5 عدد از هر کلاس

در ادامه می‌توانید تعدادی از نتایج نمایش داده شده را مشاهده نمایید.



شکل 33: تعدادی از کلاس تصاویر موجود در دیتاست

همان طور که در شکل بالا مشاهده می‌کنید، این تصاویر پیش‌زمینه‌های مختلف، شرایط نوری مختلف و زوایای مختلفی دارند و مانند دیتاست‌های MNIST نیستند که همه در شرایط تقریباً یکسانی باشند و هر کدام از تصاویر feature مخصوص به خودش را دارد و همین موضوع دسته بندی را بسیار دشوارتر می‌کند.

برای این که بتوانیم کمی فرآیند دسته‌بندی را آسان‌تر کنیم در این دیتاست باید ابتدا کار اقدام به Pre Processing کنیم تا کمی فرآیند دسته‌بندی را برای ما آسان کند.

برای قسمت اول سوال فقط از اعمال دو فیلتر ReSize و همچنین Normalize استفاده می‌کنم که فیلتر اول برای این است که سایز تصاویر که در اندازه‌های مختلفی هستند را همگی را به سایز خواسته شده در صورت سوال یعنی 30×30 تبدیل کنیم و فیلتر دوم نیز به منظور نرمال‌سازی داده‌ها است.

```

1  image_size = 30
2  tfms = transforms.Compose([
3      transforms.Resize((image_size, image_size)), # PIL Image
4      transforms.ToTensor(), # Tensor
5      transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
6  ])

```

شکل 34: اعمال Preprocessing بر روی دیتاست

مشکل بعدی در لود کردن تصاویر با استفاده از Pytorch روی می‌داد که با استفاده از دستور ImageFolder که می‌رفتم، ترتیب کلاس‌ها را به دلیل زیاد بودن آن‌ها به هم می‌ریخت و بنابراین من برای رفع این مشکل مجبور شدم که این کلاس را شخصی سازی کنم و تابع مورد نظر را اصلاح کنم و این کار را به صورت زیر با این دستورات انجام دادم:

```

1. from torchvision.datasets import ImageFolder
2. # from torchvision.datasets import VisionDataset
3.
4. class ImageFolderNew(ImageFolder):
5.     def _find_classes(self, dir):
6.         """
7.             Finds the class folders in a dataset.
8.
9.             Args:
10.                 dir (string): Root directory path.
11.
12.             Returns:
13.                 tuple: (classes, class_to_idx) where classes are relative to (dir), a
14.                     and class_to_idx is a dictionary.
15.
16.             Ensures:
17.                 No class is a subdirectory of another.
18.                 """
19.                 if sys.version_info >= (3, 5):
20.                     # Faster and available in Python 3.5 and above
21.                     classes = [d.name for d in os.scandir(dir) if d.is_dir()]
22.                 else:
23.                     classes = [d for d in os.listdir(dir) if os.path.isdir(os.path.join(d
24.                         ir, d))]
25.                     classes.sort(key=int)
26.                     class_to_idx = {classes[i]: i for i in range(len(classes))}
27.                     return classes, class_to_idx

```

پس از انجام این بخش تصاویر مربوط به مرحله train و validation را توانسته‌ایم لود کنیم. و حالا تصاویر test می‌ماند که این تصاویر چون که فرمت قرارگیری آن‌ها فرق می‌کند و لیبل آن‌ها درون فایل Test.csv می‌باشد باید با دستور متفاوتی آن‌ها را لود کنیم و به این منظور من یک کلاس مخصوص لود کردن دیتاست نوشتم:

```

1. from torch.utils.data.dataset import Dataset
2.
3. class GTSDataset(Dataset):
4.     def __init__(self, csv_file, root_dir, transform=None):

```

```

5.      """
6.      Args:
7.          csv_file (string): Path to the csv file with annotations.
8.          root_dir (string): Directory with all the images.
9.          transform (callable, optional): Optional transform to be applied
10.             on a sample.
11.      """
12.      data = pd.read_csv(csv_file)
13.      data = data.drop(columns=['Width', 'Height', 'Roi.X1', 'Roi.Y1', 'Roi.X2'
14. , 'Roi.Y2'])
15.      self.data = pd.DataFrame(data)
16.      # df_test = pd.read_csv(META_PATH + 'Test.csv')
17.      self.data['Path'] = data['Path'].str.lower()
18.      self.data['ClassId'] = data['ClassId'].apply(str)
19.
20.      self.root_dir = root_dir
21.      self.transform = transform
22.
23.      def __len__(self):
24.          return len(self.data)
25.
26.      def __getitem__(self, idx):
27.          if torch.is_tensor(idx):
28.              idx = idx.tolist()
29.          pth = self.data.iloc[idx, 1].replace("test/", "")
30.          img_name = os.path.join(self.root_dir, pth)
31.          image = io.imread(img_name)
32.          labels = self.data.at[idx, 'ClassId'] # to access a single item in Pandas
33.          DataFrame
34.          labels = np.array([labels])
35.          labels = labels.astype('long').reshape(-1, 1)
36.
37.          if self.transform:
38.              sample[0] = self.transform(sample[0])
39.          return sample

```

پس از این کار می‌رویم سراغ استفاده از این دستورات که در ادامه در این مورد توضیح می‌دهم.

```

1 image_size = 30
2
3 tfms = transforms.Compose([
4     transforms.ToPILImage(),
5     transforms.Resize((image_size, image_size)), # PIL Image
6     transforms.ToTensor(), # Tensor
7     transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
8 ])
9 test_dataset = GTSBDataset('German-Traffic-Signs-Dataset-GTSRB/Test.csv', test_dir, transform=tfms)
10
11 test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=300, shuffle=False)
12
13
14 train_dataset = ImageFolderNew(root=train_dir, transform=tfms)
15
16 number_of_train = (0.8 * len(train_dataset)).__int__()
17 number_of_valid = (0.2 * len(train_dataset)).__int__() + 1
18
19 data_train, data_valid = random_split(train_dataset, [number_of_train, number_of_valid])
20
21 train_loader = torch.utils.data.DataLoader(dataset=data_train, batch_size=batch_size, shuffle=True)
22 valid_loader = torch.utils.data.DataLoader(dataset=data_valid, batch_size=batch_size, shuffle=True)

```

شکل 35: کد مربوط به لود کردن دیتاستهای تست و تربین برای انجام Train روی آنها

در شکل بالا همان‌طور که مشاهده می‌کنید اقدام به لود کردن دیتاست می‌کنم و در قسمت مشخص شده با علامت قرمز اقدام به مشخص کردن اندازه دو مجموعه train و valid می‌کنم. به این صورت که 80 درصد مجموعه train مربوط به داده‌های Training و 20 درصد باقی‌مانده مربوط به داده‌های Validation است.

جدول 2: تعداد داده‌ها در هر مجموعه

| نام مجموعه | تعداد نمونه |
|------------|--------------|
| Training | 31376 |
| Validation | 7842 |
| Testing | 12630 |

همچنین من کل داده‌های موجود در پوشه Train (داده‌های مجموعه تقسیم شده Validation) را نیز نمودار پراکندگی اعضای آن را نسبت به هر کلاس با استفاده از دستور زیر رسم کردم:

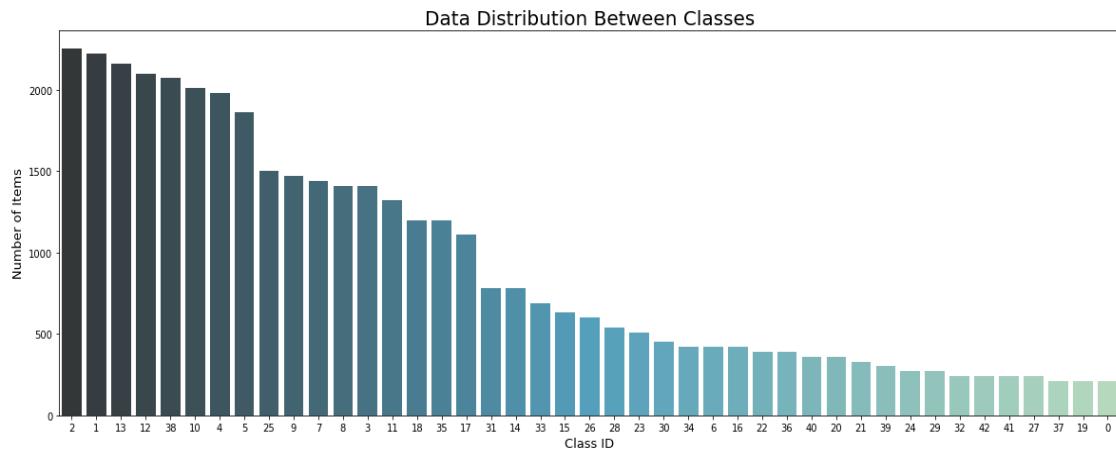
```

1 import seaborn as sns
2
3 f, ax = plt.subplots(1, 1, figsize=(20, 7))
4 sns.countplot(x='ClassId', data=data, ax=ax, order=data.ClassId.value_counts().index, palette="GnBu_d")
5 plt.show()

```

شکل 36: دستور رسم نمودار

که پس از اجرای آن نمودار به شکل زیر می‌شود:



شکل 37: نمودار تعداد داده‌ها در هر کلاس در فولدر Train

همان‌طور که از شکل بالا مشخص است کلاس‌های مختلف تعداد داده‌های بسیار متفاوتی دارند و این می‌تواند در مواردی مشکل‌ساز شود و روی دقت ما تاثیر منفی بگذارد اما چون تعداد کلاس‌های ما زیاد است این تاثیر خیلی به چشم نمی‌آید و بیشتر در دیتاست‌های با تعداد کلاس‌های کم این موضوع خود را نشان می‌دهد.

پس از انجام تمامی این کارها اقدام به طراحی شبکه مربوطه برای آموزش این دیتاست می‌کنم از ما در سوال خواسته شده است که شبکه CNN با سه لایه را طراحی کنیم.

قسمت A

در ابتدا مشخصات لایه‌های CNN را قرار می‌دهم:

جدول 3: مشخصات سه لایه کانولوشنی مورد استفاده در سوال

| تعداد فیلتر | Activation Function | Padding | Stride | اندازه پنجره Kernel | نوع | شماره لایه |
|-------------|---------------------|---------|--------|---------------------|----------------------|------------|
| 73 | ReLU | 0 | 1 | 5 | CNN | لایه اول |
| - | - | - | - | - | Max Pooling (نداریم) | |
| 153 | ReLU | 0 | 1 | 3 | CNN | لایه دوم |
| - | - | - | 2 | 2 | Max Pooling | |
| 273 | ReLU | 1 | 1 | 3 | CNN | لایه سوم |
| - | - | - | 2 | 2 | Max Pooling | |

لایه‌های Fully Connected ✓

جدول 4: مشخصات لایه‌های آخر شبکه Fully Connected

| نام لایه | تعداد نرون‌ها | تابع Activation |
|------------------|---------------|-----------------|
| Fully Connected1 | 500 | ReLU |
| Fully Connected2 | 43 | LogSigmoid |

✓ مشخصات کلی شبکه:

جدول ۵: مشخصات کلی شبکه

| | |
|------------------|-----------------------|
| CrossEntropyLoss | :Loss Function |
| Adam | :Optimizer |
| 300 | :Batch Size |
| 30 | :Epochs |

لازم به ذکر است که در Pytorch هنگامی که از CrossEntropyLoss .Loss Function استفاده می‌کنیم طبق داکیومنت خود پایتورچ^۱ (ذکر شده در پایین صفحه) اقدام به اعمال LogSoftmax در لایه آخر می‌کند به همین دلیل من در لایه آخر چنانچه که در کد ببینید این گزینه را comment کرده‌ام. در ادامه می‌توانید کد مربوط به مدل طراحی شده برای این قسمت از سوال را مشاهده نمایید:

<https://pytorch.org/docs/stable/nn.html#crossentropyloss>¹

```

1  class CnnModel(nn.Module):
2      def __init__(self):
3          super(CnnModel, self).__init__()
4          self.layer1 = nn.Sequential(
5              nn.Conv2d(in_channels=3, out_channels=73, kernel_size=5, stride=1, padding=0),
6              nn.ReLU(),
7          )
8          self.layer2 = nn.Sequential(
9              nn.Conv2d(in_channels=73, out_channels=153, kernel_size=3, stride=1, padding=0),
10             nn.ReLU(),
11             nn.MaxPool2d(kernel_size=2, stride=2),
12         )
13         self.layer3 = nn.Sequential(
14             nn.Conv2d(in_channels=153, out_channels=273, kernel_size=3, stride=1, padding=1),
15             nn.ReLU(),
16             nn.MaxPool2d(kernel_size=2, stride=2),
17         )
18         self.fc1 = nn.Linear(6 * 6 * 273, 500)
19         self.relu = nn.ReLU()
20         self.fc2 = nn.Linear(500, 43)
21         self.softmax = nn.LogSoftmax(dim=1)
22
23     def forward(self, x):
24         out = self.layer1(x)
25         out = self.layer2(out)
26         out = self.layer3(out)
27
28         out = torch.flatten(out, 1)
29         out = self.fc1(out)
30         out = self.relu(out)
31         out = self.fc2(out)
32         # out = self.softmax(out)
33         return out

```

شکل 38: کد مربوط به Model طراحی شده برای شبکه عصبی

همچنین در ادامه توابع Loss و Optimizer را می‌توانید مشاهده نمایید:

```

1  from torch.optim.lr_scheduler import MultiStepLR
2  learning_rate = 0.00073
3  epochs = 30
4
5  model = CnnModel().to(CUDA)
6
7  criterion = nn.CrossEntropyLoss()
8  optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
9
10 scheduler = MultiStepLR(optimizer, milestones=[50, 70, 90], gamma=0.7)

```

شکل 39: کد مربوط به Optimizer و Learning Rate و ... در پایتورج

قسمت مشخص شده در شکل بالا دستوری است در پایتورج که در آن می‌توانید مشخص کنید که در های مشخص شده در قسمت Learning Rate باید و با ضریب gamma مقدار Epoch را تغییر دهد.

در زیر هم می‌توانید کد مربوط به فرآیند آموزش شبکه را مشاهده کنید:

```
1. import time
2. start_time = time.time()
3. running_loss_history = []
4. running_correct_history = []
5. val_running_loss_history = []
6. val_running_corrects_history = []
7.
8. for e in range(epochs):
9.     running_loss = 0.0
10.    running_corrects = 0.0
11.    val_running_loss = 0.0
12.    val_running_corrects = 0.0
13.    for inputs, labels in train_loader:
14.        if torch.cuda.is_available():
15.            inputs = inputs.to(CUDA)
16.            labels = labels.to(CUDA)
17.
18.        outputs = model(inputs)
19.        loss = criterion(outputs, labels)
20.        optimizer.zero_grad()
21.        loss.backward()
22.        optimizer.step()
23.
24.
25.        _, preds = torch.max(outputs, 1)
26.
27.        running_loss += loss.item()
28.        running_corrects += torch.sum(preds == labels.data) # it will give us the
   number of correct prediction for a single batch of images...
29.
30.    else:
31.        with torch.no_grad(): # it will temporarily set all the required grad f
   lags to be false
32.        for val_inputs, val_labels in valid_loader:
33.            if torch.cuda.is_available():
34.                val_labels = val_labels.to(CUDA)
35.                val_inputs = val_inputs.to(CUDA)
36.
37.            val_outputs = model(val_inputs)
38.            val_loss = criterion(val_outputs, val_labels)
39.
40.            _, val_preds = torch.max(val_outputs, 1)
41.            val_running_loss += val_loss.item()
42.            val_running_corrects += torch.sum(val_preds == val_labels.data)
43.
44.    epoch_loss = running_loss / len(train_loader.dataset)
45.    epoch_acc = running_corrects.float() / len(data_train)
46.    running_loss_history.append(epoch_loss)
47.    running_correct_history.append(epoch_acc)
48.
49.    val_epoch_loss = val_running_loss / len(valid_loader.dataset)
50.    val_epoch_acc = val_running_corrects.float() / len(data_valid)
51.    val_running_loss_history.append(val_epoch_loss)
52.    val_running_corrects_history.append(val_epoch_acc)
53.    # if (e + 1) % 10 == 0:
54.    print("Epoch : ", (e+1))
55.    print(' Training Loss=> {:.8f}, Training ACC {:.4f} '.format(epoch_l
   oss, epoch_acc.item()))
56.    print('Validation Loss=> {:.8f}, Validation ACC {:.4f} '.format(val_epo
   ch_loss, val_epoch_acc.item()))
57.    scheduler.step()
```

```

58.     for param_group in optimizer.param_groups:
59.         print("*** Learning Rate : => {:.9f}".format(param_group['lr']))
60.         print("---->>>-----<<<-***")
61.
62. print("*****")
63. print("---->>> %s seconds <<<---" % (time.time() - start_time))
64. print("*****")

```

قسمت B

در این قسمت از ما خواسته شده است که نتایج اجرای شبکه معرفی شده در قسمت قبل را قرار دهیم.

در این قسمت من به همان 30 ایپاک گفته شده کفایت کردم اما می‌شد با کمی ادامه دادن به دقت‌های بهتری هم رسید اما چون با همین مقدار هم من به دقت قابل قبولی رسیدم به نظرم کافی بود.

در ادامه می‌توانید مقادیر Loss و Accuracy مربوط به 5 ایپاک اول را مشاهده نمایید:

```

Epoch : 1
    Training Loss:=> 0.00469361,      Training ACC 0.6091
Validation Loss:=> 0.00090094, Validation ACC 0.9267
*** Learning Rate : => 0.000730000
***->>>-----<<<-***

Epoch : 2
    Training Loss:=> 0.00043948,      Training ACC 0.9642
Validation Loss:=> 0.00033807, Validation ACC 0.9730
*** Learning Rate : => 0.000730000
***->>>-----<<<-***

Epoch : 3
    Training Loss:=> 0.00015143,      Training ACC 0.9891
Validation Loss:=> 0.00023250, Validation ACC 0.9828
*** Learning Rate : => 0.000730000
***->>>-----<<<-***

Epoch : 4
    Training Loss:=> 0.00006655,      Training ACC 0.9956
Validation Loss:=> 0.00015324, Validation ACC 0.9921
*** Learning Rate : => 0.000730000
***->>>-----<<<-***

Epoch : 5
    Training Loss:=> 0.00006804,      Training ACC 0.9948
Validation Loss:=> 0.00016601, Validation ACC 0.9901
*** Learning Rate : => 0.000730000
***->>>-----<<<-***

```

شکل 40: 5 ایپاک اول مربوط به اجرای آموزش

در ادامه نیز می‌توانید نتایج مربوط به 5 ایپاک آخر را مشاهده نمایید:

```

****->>>-----<<<-***  

Epoch : 25  

    Training Loss=> 0.00000030,      Training ACC 1.0000  

Validation Loss=> 0.00015744, Validation ACC 0.9963  

*** Learning Rate : => 0.000730000  

***->>>-----<<<-***  

Epoch : 26  

    Training Loss=> 0.00000009,      Training ACC 1.0000  

Validation Loss=> 0.00016226, Validation ACC 0.9963  

*** Learning Rate : => 0.000730000  

***->>>-----<<<-***  

Epoch : 27  

    Training Loss=> 0.00000004,      Training ACC 1.0000  

Validation Loss=> 0.00016181, Validation ACC 0.9964  

*** Learning Rate : => 0.000730000  

***->>>-----<<<-***  

Epoch : 28  

    Training Loss=> 0.00000003,      Training ACC 1.0000  

Validation Loss=> 0.00016367, Validation ACC 0.9963  

*** Learning Rate : => 0.000730000  

***->>>-----<<<-***  

Epoch : 29  

    Training Loss=> 0.00000003,      Training ACC 1.0000  

Validation Loss=> 0.00016507, Validation ACC 0.9964  

*** Learning Rate : => 0.000730000  

***->>>-----<<<-***  

Epoch : 30  

    Training Loss=> 0.00000002,      Training ACC 1.0000  

Validation Loss=> 0.00016654, Validation ACC 0.9966  

*** Learning Rate : => 0.000730000  

***->>>-----<<<-***
```

شکل 41: نتایج به دست آمده مربوط به 5 ایپاک آخر اجرای فرآیند آموزش

همچنین مدت زمان اجرای فرآیند آموزش برای این 30 ایپاک نیز به شکل زیر است:

```

*****  

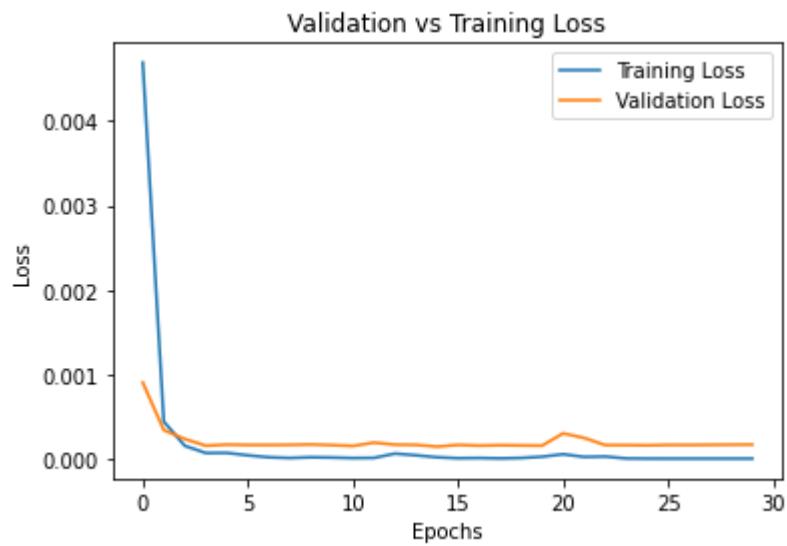
--->>> 629.1948792934418 seconds <<<---  

*****
```

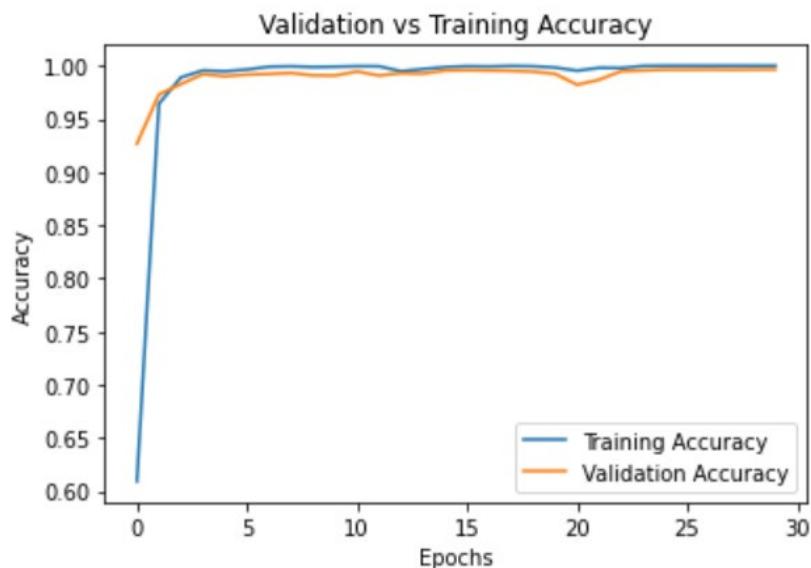
شکل 42: مدت زمان اجرای فرآیند آموزش

در ادامه نمودارهای به دست آمده را قرار می‌دهم.

در زیر می‌توانید نمودار مقایسه Loss را برای دو مجموعه Train و Validation مشاهده نمایید:



شکل 43: نمودار مقایسه Loss در دو مجموعه داده Train و Validation



شکل 44: نمودار مقایسه Accuracy در دو مجموعه داده Train و Validation

همان‌طور که در دو نمودار بالا مشخص است و همین‌طور در شکلی که از 5 ایپاک آخر داده‌های Train قرار دادم نیز مشخص است در این داده‌ها توانسته‌ام به دقت‌های بسیار خوبی برسم و در داده‌های Train توانسته‌ام حتی به دقت 100 درصدی برسم و در داده‌های Validation نیز به دقت حدود 99.7 درصد رسیده‌ام که بسیار مناسب است.

و سپس با کد زیر اقدام به آزمایش دقت شبکه بر روی داده‌های تست کردم:

```

1. test_running_loss_history = []
2. test_running_corrects_history = []
3. test_running_loss = 0.0
4. test_running_corrects = 0.0
5.
6. for test_inputs, test_labels in test_loader:
7.     # print(test_inputs.shape)
8.     test_labels = test_labels.view(test_labels.shape[0])
9.     # print(test_labels.shape)
10.    if torch.cuda.is_available():
11.        test_labels = test_labels.to(CUDA)
12.        test_inputs = test_inputs.to(CUDA)
13.
14.
15.    test_outputs = model(test_inputs)
16.    test_loss = criterion(test_outputs, test_labels)
17.
18.    _, test_preds = torch.max(test_outputs, 1)
19.    test_running_corrects += torch.sum(test_preds == test_labels.data)
20.    test_running_loss += test_loss.item()
21.
22.
23. test_epoch_acc = test_running_corrects.float() / len(test_dataset)
24. test_epoch_loss = test_running_loss / len(test_loader.dataset)
25. print('Test Loss: {:.8f}, Test ACC: {:.4f}'.format(test_epoch_loss, test_epoch_
acc.item()))

```

Test Loss: 0.0010, Test ACC: 0.9614

شکل 45: دقت به دست آمده بر روی داده‌های Test

همان طور که در شکل بالا می‌توانید مشاهده کنید در داده‌های Test توانسته‌ام که به دقت بسیار خوبی حدود 96٪ برسم که دقت مناسبی هست و مقدار Loss به حدود 0.0010 رسیده است که مقدار کم و مناسبی هست. و حتی می‌شد با آموزش بیشتر شبکه به دقت‌های بیشتری هم رسید.

❖ قسمت C :

در این قسمت از ما خواسته شده است که اقدام به رسم ماتریس آشفتگی مربوط به نتایج به دست آمده از قسمت قبل کنیم.

می‌توانید کد رسم ماتریس را در ادامه مشاهده کنید:

```

1. from sklearn.metrics import confusion_matrix
2. import itertools
3. from tqdm import tqdm
4. from torch.autograd import Variable

```

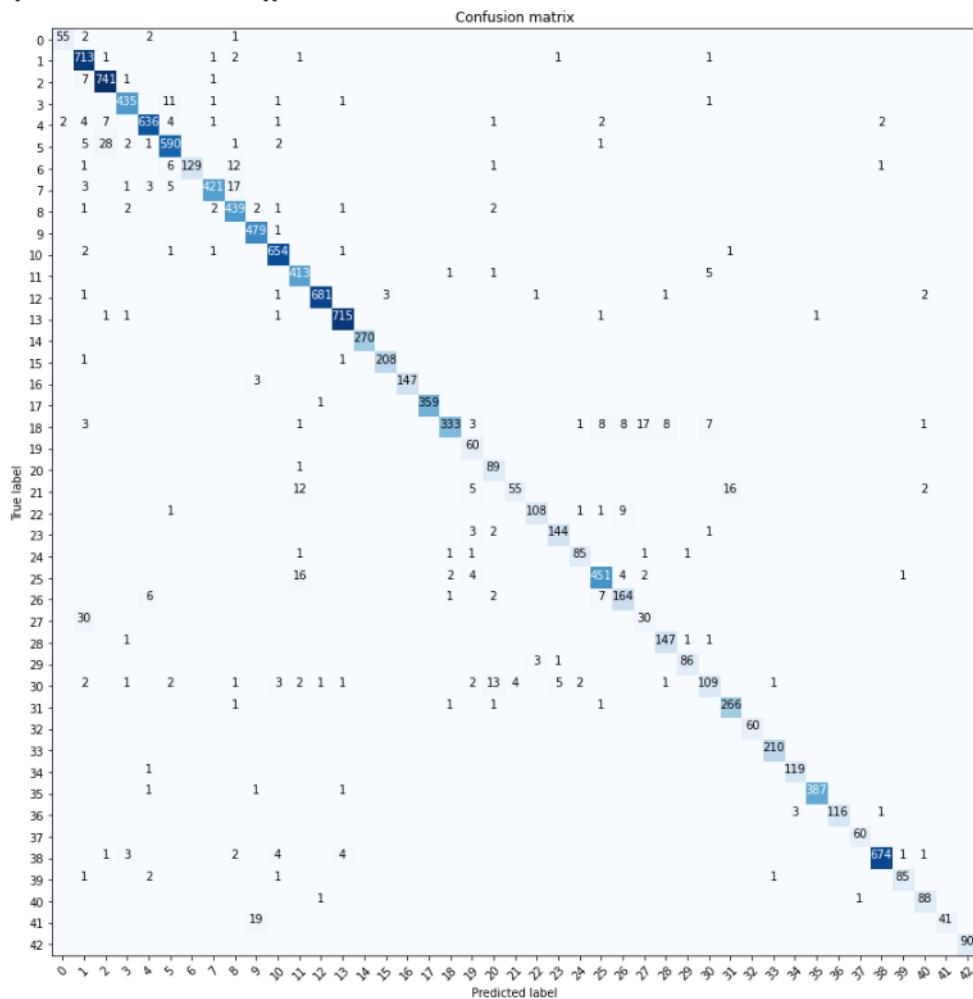
```

5.
6.
7. def plot_confusion_matrix(cm, classes, normalize=False, figsize=(12, 12), title='Confusion matrix', cmap=plt.cm.Blues):
8.     """
9.     This function prints and plots the confusion matrix.
10.    Normalization can be applied by setting `normalize=True`.
11.    (This function is copied from the scikit docs.)
12.    """
13.    plt.figure(figsize=figsize)
14.    plt.imshow(cm, interpolation='nearest', cmap=cmap)
15.    plt.title(title)
16.    plt.colorbar()
17.    tick_marks = np.arange(len(classes))
18.    plt.xticks(tick_marks, classes, rotation=45)
19.    plt.yticks(tick_marks, classes)
20.
21.    if normalize: cm = cm.astype('int') / cm.sum(axis=1)[:, np.newaxis]
22.    print(cm)
23.    thresh = cm.max() / 2.
24.    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
25.        annot = "%d" % cm[i, j] if cm[i, j] > 0 else ""
26.        plt.text(int(j), int(i), annot, horizontalalignment="center", color="white" if cm[i, j] > thresh else "black")
27.
28.    plt.tight_layout()
29.    plt.ylabel('True label')
30.    plt.xlabel('Predicted label')
31.
32. def to_var(x, volatile=False):
33.     if torch.cuda.is_available():
34.         x = x.cuda()
35.     return Variable(x, volatile=volatile)
36.
37. def predict_class(model, dataloader):
38.     """ Predict probabilities for the given model and dataset
39.     """
40.     model.train(False)
41.     result = []
42.     y = []
43.
44.     for inputs, targets in tqdm(dataloader):
45.         targets = targets.view(targets.shape[0])
46.         if torch.cuda.is_available():
47.             targets = targets.to(CUDA)
48.             inputs = inputs.to(CUDA)
49.         scores = model(inputs)
50.         _, preds = torch.max(scores.data, 1)
51.         result += [preds.cpu().numpy()]
52.         y += [targets.cpu().numpy()]
53.
54.     result = np.concatenate(result)
55.     y = np.concatenate(y)
56.     return result, y
57.
58. y_pred, y_true = predict_class(model, test_loader)
59. cm = confusion_matrix(y_true, y_pred)
60. plot_confusion_matrix(cm, train_dataset.classes, normalize=False, figsize=(15, 12))

```

قبل از رسم نتیجه ماتریس لازم است ذکر کنم به دلیل زیاد بودن تعداد کلاس‌ها ماتریس بزرگ می‌شود و نسخه‌ای که من در اینجا قرار می‌دهم ممکن است که کمی ناخوانا باشد اما در کدی که قرار می‌دهد نسته بزرگ‌تر آن وجود دارد که خوانایی بهتری دارد.

اما در همین شکل نیز مشخص است که شبکه طراحی شده توانسته است که اکثر کلاس‌ها را درست تشخیص دهد این موضوع را می‌توانید با مشاهده رنگ‌ها بیابید.



شکل 46: ماتریس آشفتگی برای این قسمت سوال

همان‌طور که در شکل بالا مشخص است، اکثر نقاط مشخص شده روی قطر پررنگ هستند و این نقاط را در سایر جاها مشاهده نمی‌کنیم و همین امر نشان می‌دهد که اکثر داده‌ها درست کلاس بندی شده‌اند و تعداد بسیار کمی خطأ داشته‌ایم.

➤ **توجه:** تمامی کدهای این قسمت رسم ماتریس آشфтگی و همین طور دو قسمت A و B در فایل Practical_Part_A&B&C_CrossEntropyLoss.ipynb قرار دارد.

❖ قسمت D ❖

در این قسمت از ما خواسته شده است که اقدام به مقایسه عملکرد مدل طراحی شده در قسمتهای Fully Connected Activation Functionها در قسمتهای مختلف به جز لایه آخر مربوط به قبل با تغییر Activation Function بکنیم.

همچنین از ما خواسته شده است که این کار را برای سه مختلف Activation Function: Tanh, ReLU و Sigmoid انجام دهیم.

در ابتدا مدل طراحی شده برای ReLU را در ادامه قرار می‌دهم:

```

35  class CnnModelReLU(nn.Module):
36      def __init__(self):
37          super(CnnModelReLU, self).__init__()
38          self.layer1 = nn.Sequential(
39              nn.Conv2d(in_channels=3, out_channels=73, kernel_size=5, stride=1, padding=0),
40              nn.ReLU(),
41          )
42          self.layer2 = nn.Sequential(
43              nn.Conv2d(in_channels=73, out_channels=153, kernel_size=3, stride=1, padding=0),
44              nn.ReLU(),
45              nn.MaxPool2d(kernel_size=2, stride=2),
46          )
47          self.layer3 = nn.Sequential(
48              nn.Conv2d(in_channels=153, out_channels=273, kernel_size=3, stride=1, padding=1),
49              nn.ReLU(),
50              nn.MaxPool2d(kernel_size=2, stride=2),
51          )
52          self.fc1 = nn.Linear(6 * 6 * 273, 500) # 1: Stride
53          self.relu = nn.ReLU()
54          self.fc2 = nn.Linear(500, 43) # 1: Stride
55          self.softmax = nn.LogSoftmax(dim=1)
56
57      def forward(self, x):
58          out = self.layer1(x)
59          out = self.layer2(out)
60          out = self.layer3(out)
61
62          out = torch.flatten(out, 1)
63          out = self.fc1(out)
64          out = self.relu(out)
65          out = self.fc2(out)
66          # out = self.softmax(out)
67          return out

```

شکل 47: مدل طراحی شده برای ReLU

که مدلی که در شکل بالا مشاهده می‌کنید مشابه مدل قسمت‌های A و B می‌باشد.

در ادامه می‌توانید مدل مربوط به Tanh را مشاهده کنید:

```
70 class CnnModelTanh(nn.Module):
71     def __init__(self):
72         super(CnnModelTanh, self).__init__()
73         self.layer1 = nn.Sequential(
74             nn.Conv2d(in_channels=3, out_channels=73, kernel_size=5, stride=1, padding=0),
75             nn.Tanh(),
76         )
77         self.layer2 = nn.Sequential(
78             nn.Conv2d(in_channels=73, out_channels=153, kernel_size=3, stride=1, padding=0),
79             nn.Tanh(),
80             nn.MaxPool2d(kernel_size=2, stride=2),
81         )
82         self.layer3 = nn.Sequential(
83             nn.Conv2d(in_channels=153, out_channels=273, kernel_size=3, stride=1, padding=1),
84             nn.Tanh(),
85             nn.MaxPool2d(kernel_size=2, stride=2),
86         )
87         self.fc1 = nn.Linear(6 * 6 * 273, 500) # 1: Stride
88         self.tanh = nn.Tanh()
89         self.fc2 = nn.Linear(500, 43) # 1: Stride
90         # self.softmax = nn.LogSoftmax(dim=1)
91
92     def forward(self, x):
93         out = self.layer1(x)
94         out = self.layer2(out)
95         out = self.layer3(out)
96
97         out = torch.flatten(out, 1)
98         out = self.fc1(out)
99         out = self.tanh(out)
100        out = self.fc2(out)
101        # out = self.softmax(out)
102        return out
```

شکل 48: مدل طراحی شده برای Tanh

سپس در انتهای نیز می‌توانید مدل مربوط به Sigmoid را مشاهده نمایید:

```

104 class CnnModelSigmoid(nn.Module):
105     def __init__(self):
106         super(CnnModelSigmoid, self).__init__()
107         self.layer1 = nn.Sequential(
108             nn.Conv2d(in_channels=3, out_channels=73, kernel_size=5, stride=1, padding=0),
109             nn.Sigmoid(),
110         )
111         self.layer2 = nn.Sequential(
112             nn.Conv2d(in_channels=73, out_channels=153, kernel_size=3, stride=1, padding=0),
113             nn.Sigmoid(),
114             nn.MaxPool2d(kernel_size=2, stride=2),
115         )
116         self.layer3 = nn.Sequential(
117             nn.Conv2d(in_channels=153, out_channels=273, kernel_size=3, stride=1, padding=1),
118             nn.Sigmoid(),
119             nn.MaxPool2d(kernel_size=2, stride=2),
120         )
121         self.fc1 = nn.Linear(6 * 6 * 273, 500) # 1: Stride
122         self.sigmoid = nn.Sigmoid()
123         self.fc2 = nn.Linear(500, 43) # 1: Stride
124         self.softmax = nn.LogSoftmax(dim=1)
125
126     def forward(self, x):
127         out = self.layer1(x)
128         out = self.layer2(out)
129         out = self.layer3(out)
130
131         out = torch.flatten(out, 1)
132         out = self.fc1(out)
133         out = self.sigmoid(out)
134         out = self.fc2(out)
135         # out = self.softmax(out)
136         return out

```

شکل 49: مدل طراحی شده برای Sigmoid

همان‌طور که در شکل‌های بالا می‌توانید مشاهده کنید، تمامی این مدل‌ها فقط function‌های آن‌ها تغییر کرده است و سایر موارد مشابه هم هستند.

همچنین لازم به ذکر است که من برای این قسمت فرآیند Train را تبدیل به یک تابع مانند زیر کردم:

```

1. import time
2.
3. def model_training(model=None, epochs=15, optimizer=None, scheduler=None, criteri
on=nn.CrossEntropyLoss()):
4.     start_time = time.time()
5.     running_loss_history = []
6.     running_correct_history = []
7.     val_running_loss_history = []
8.     val_running_corrects_history = []
9.
10.    for e in range(epochs):
11.        running_loss = 0.0
12.        running_corrects = 0.0
13.        val_running_loss = 0.0
14.        val_running_corrects = 0.0
15.        for inputs, labels in train_loader:
16.            if torch.cuda.is_available():
17.                inputs = inputs.to(CUDA)

```

```

18.         labels = labels.to(CUDA)
19.
20.         outputs = model(inputs)
21.         loss = criterion(outputs, labels)
22.         optimizer.zero_grad()
23.         loss.backward()
24.         optimizer.step()
25.
26.
27.         _, preds = torch.max(outputs, 1)
28.
29.         running_loss += loss.item()
30.         running_corrects += torch.sum(preds == labels.data) # it will give us
   the number of correct prediction for a single batch of images...
31.
32.     else:
33.         with torch.no_grad(): # it will temporarorly set all the required gr
   ad flags to be false
34.             for val_inputs, val_labels in valid_loader:
35.                 if torch.cuda.is_available():
36.                     val_labels = val_labels.to(CUDA)
37.                     val_inputs = val_inputs.to(CUDA)
38.
39.                     val_outputs = model(val_inputs)
40.                     val_loss = criterion(val_outputs, val_labels)
41.
42.                     _, val_preds = torch.max(val_outputs, 1)
43.                     val_running_loss += val_loss.item()
44.                     val_running_corrects += torch.sum(val_preds == val_labels.dat
   a)
45.
46.             epoch_loss = running_loss / len(train_loader.dataset)
47.             epoch_acc = running_corrects.float() / len(data_train)
48.             running_loss_history.append(epoch_loss)
49.             running_correct_history.append(epoch_acc)
50.
51.             val_epoch_loss = val_running_loss / len(valid_loader.dataset)
52.             val_epoch_acc = val_running_corrects.float() / len(data_valid)
53.             val_running_loss_history.append(val_epoch_loss)
54.             val_running_corrects_history.append(val_epoch_acc)
55.             # if (e + 1) % 10 == 0:
56.             print("Epoch : ", (e+1))
57.             print(' Training Loss=> {:.8f}, Training ACC {:.4f} '.format(epo
   ch_loss, epoch_acc.item()))
58.             print('Validation Loss=> {:.8f}, Validation ACC {:.4f} '.format(val
   _epoch_loss, val_epoch_acc.item()))
59.             if scheduler is not None:
60.                 scheduler.step()
61.                 for param_group in optimizer.param_groups:
62.                     print("''' Learning Rate : => {:.9f}'.format(param_group['lr'
   ])) 
63.                     print("'''->>-----<<-
   ***")
64.
65.             print("*****")
66.             print("---->> %s seconds <<---" % (time.time() - start_time))
67.             print("*****")
68.             return running_loss_history, val_running_loss_history, running_correct_hist
   ory, val_running_corrects_history

```

همان طور که در بالا نیز می‌توانید مشاهده کنید من برای این تابع پارامترهایی مانند تعداد epoch و تابع Loss و مدل طراحی شده را می‌فرستم و سپس این تابع اقدام به آموزش مدلی که برای آن فرستاده‌ام می‌کند.

در زیر نیز من این پارامترها را مشخص می‌کنم و برای تابع معرفی شده ارسال می‌کنم:

```
1. from torch.optim.lr_scheduler import MultiStepLR
2.
3.
4. learning_rate = 0.00073
5. epochs = 15
6.
7. criterion = nn.CrossEntropyLoss()
8.
9. # scheduler_relu = MultiStepLR(optimizer_relu, milestones=[30, 70], gamma=0.1)
10.
11. model_relu = CnnModelReLU().to(CUDA)
12. optimizer_relu = torch.optim.Adam(model_relu.parameters(), lr=learning_rate)
13. print("### *** ReLU Optimizer : ===>> *** ###")
14. train_loss_relu, val_loss_relu, train_acc_relu, val_acc_relu = model_training(model=model_relu, epochs=epochs, optimizer=optimizer_relu, scheduler=None, criterion=criterion)
15. #####
16. model_sigmoid = CnnModelSigmoid().to(CUDA)
17. optimizer_sigmoid = torch.optim.Adam(model_sigmoid.parameters(), lr=learning_rate)
18.
19. print("### *** Sigmoid Optimizer : ===>> *** ###")
20. train_loss_sigmoid, val_loss_sigmoid, train_acc_sigmoid, val_acc_sigmoid = model_training(model=model_sigmoid, epochs=epochs, optimizer=optimizer_sigmoid, scheduler=None, criterion=criterion)
21. #####
22. model_tanh = CnnModelTanh().to(CUDA)
23. optimizer_tanh = torch.optim.Adam(model_tanh.parameters(), lr=learning_rate)
24. print("### *** Tanh Optimizer : ===>> *** ###")
25. train_loss_tanh, val_loss_tanh, train_acc_tanh, val_acc_tanh = model_training(model=model_tanh, epochs=epochs, optimizer=optimizer_tanh, scheduler=None, criterion=criterion)
26. #####
27. model_leaky = CnnModelLeakyReLU().to(CUDA)
28. optimizer_leaky = torch.optim.Adam(model_leaky.parameters(), lr=learning_rate)
29. print("### *** Leaky ReLU Optimizer : ===>> *** ###")
30. train_loss_leaky, val_loss_leaky, train_acc_leaky, val_acc_leaky = model_training(model=model_leaky, epochs=epochs, optimizer=optimizer_leaky, scheduler=None, criterion=criterion)
```

در ادامه من مقادیر به دست آمده برای 5 ایپاک اول و آخر این سه مدل را قرار می‌دهم:

```

### *** ReLU Activation Function : ==>> *** ##
Epoch : 1
    Training Loss=> 0.00433762,    Training ACC 0.6477
Validation Loss=> 0.00086148, Validation ACC 0.9277
***->>>-----<<<-***
Epoch : 2
    Training Loss=> 0.00041576,    Training ACC 0.9669
Validation Loss=> 0.00034292, Validation ACC 0.9755
***->>>-----<<<-***
Epoch : 3
    Training Loss=> 0.00014672,    Training ACC 0.9896
Validation Loss=> 0.00021894, Validation ACC 0.9874
***->>>-----<<<-***
Epoch : 4
    Training Loss=> 0.00008607,    Training ACC 0.9931
Validation Loss=> 0.00020930, Validation ACC 0.9856
***->>>-----<<<-***
Epoch : 5
    Training Loss=> 0.00005095,    Training ACC 0.9962
Validation Loss=> 0.00015980, Validation ACC 0.9917
***->>>-----<<<-***

```

شکل ۵۰: اجرای ۱۵ ایپاک اول ReLU

```

***->>>-----<<<-***
Epoch : 10
    Training Loss=> 0.00001790,    Training ACC 0.9986
Validation Loss=> 0.00013015, Validation ACC 0.9926
***->>>-----<<<-***
Epoch : 11
    Training Loss=> 0.00004586,    Training ACC 0.9969
Validation Loss=> 0.00012424, Validation ACC 0.9936
***->>>-----<<<-***
Epoch : 12
    Training Loss=> 0.00001245,    Training ACC 0.9990
Validation Loss=> 0.00012209, Validation ACC 0.9939
***->>>-----<<<-***
Epoch : 13
    Training Loss=> 0.00000179,    Training ACC 0.9999
Validation Loss=> 0.00012249, Validation ACC 0.9959
***->>>-----<<<-***
Epoch : 14
    Training Loss=> 0.00000236,    Training ACC 0.9998
Validation Loss=> 0.00013991, Validation ACC 0.9945
***->>>-----<<<-***
Epoch : 15
    Training Loss=> 0.00005018,    Training ACC 0.9960
Validation Loss=> 0.00013880, Validation ACC 0.9915
***->>>-----<<<-***
*****
```

شکل ۵۱: اجرای ۱۵ ایپاک آخر ReLU

```

### *** Sigmoid Activation Function : ==>> *** ####
Epoch : 1
    Training Loss:=> 0.01174610,    Training ACC 0.0529
Validation Loss:=> 0.01204795, Validation ACC 0.0585
***->>>-----<<<-***

Epoch : 2
    Training Loss:=> 0.01171624,    Training ACC 0.0550
Validation Loss:=> 0.01203947, Validation ACC 0.0530
***->>>-----<<<-***

Epoch : 3
    Training Loss:=> 0.01171747,    Training ACC 0.0553
Validation Loss:=> 0.01203824, Validation ACC 0.0530
***->>>-----<<<-***

Epoch : 4
    Training Loss:=> 0.01171483,    Training ACC 0.0537
Validation Loss:=> 0.01201040, Validation ACC 0.0485
***->>>-----<<<-***

Epoch : 5
    Training Loss:=> 0.01171130,    Training ACC 0.0544
Validation Loss:=> 0.01204087, Validation ACC 0.0530
***->>>-----<<<-***
```

شکل 52: اجرای 5 ایپاک اول Sigmoid

```

***->>>-----<<<-***
Epoch : 10
    Training Loss:=> 0.00112272,    Training ACC 0.9232
Validation Loss:=> 0.00089357, Validation ACC 0.9411
***->>>-----<<<-***

Epoch : 11
    Training Loss:=> 0.00058817,    Training ACC 0.9651
Validation Loss:=> 0.00058696, Validation ACC 0.9658
***->>>-----<<<-***

Epoch : 12
    Training Loss:=> 0.00036003,    Training ACC 0.9827
Validation Loss:=> 0.00044283, Validation ACC 0.9735
***->>>-----<<<-***

Epoch : 13
    Training Loss:=> 0.00025122,    Training ACC 0.9895
Validation Loss:=> 0.00033912, Validation ACC 0.9800
***->>>-----<<<-***

Epoch : 14
    Training Loss:=> 0.00017811,    Training ACC 0.9936
Validation Loss:=> 0.00031441, Validation ACC 0.9811
***->>>-----<<<-***

Epoch : 15
    Training Loss:=> 0.00012771,    Training ACC 0.9963
Validation Loss:=> 0.00024143, Validation ACC 0.9837
***->>>-----<<<-***
```

شکل 53: اجرای 5 ایپاک آخر Sigmoid

```

### *** Tanh Activation Function : ===>> *** ##
Epoch : 1
    Training Loss=> 0.00292443,    Training ACC 0.7782
Validation Loss=> 0.00047791, Validation ACC 0.9756
***->>>-----<<<-***
Epoch : 2
    Training Loss=> 0.00022103,    Training ACC 0.9892
Validation Loss=> 0.00018018, Validation ACC 0.9916
***->>>-----<<<-***
Epoch : 3
    Training Loss=> 0.00007018,    Training ACC 0.9977
Validation Loss=> 0.00011686, Validation ACC 0.9943
***->>>-----<<<-***
Epoch : 4
    Training Loss=> 0.00002633,    Training ACC 0.9996
Validation Loss=> 0.00008815, Validation ACC 0.9957
***->>>-----<<<-***
Epoch : 5
    Training Loss=> 0.00001194,    Training ACC 0.9999
Validation Loss=> 0.00008872, Validation ACC 0.9964
***->>>-----<<<-***

```

شکل 54: اجرای 5 ایپاک اول Tanh

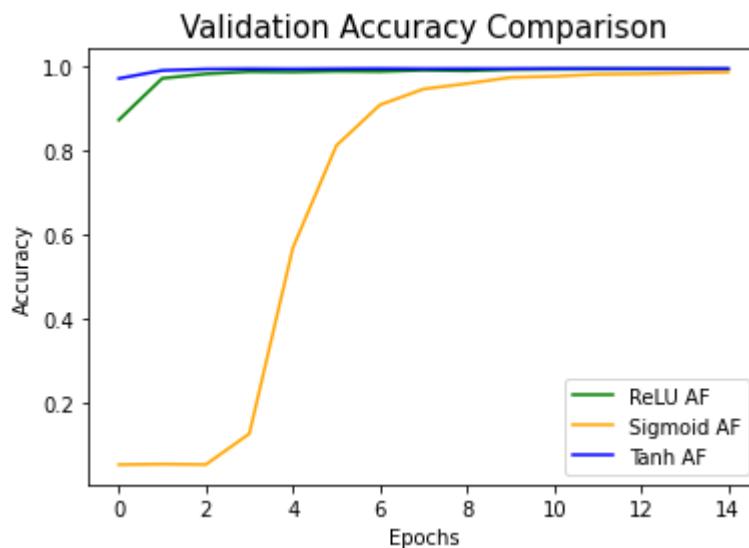
```

***->>>-----<<<-***
Epoch : 10
    Training Loss=> 0.00000261,    Training ACC 1.0000
Validation Loss=> 0.00009953, Validation ACC 0.9966
***->>>-----<<<-***
Epoch : 11
    Training Loss=> 0.00000213,    Training ACC 1.0000
Validation Loss=> 0.00006662, Validation ACC 0.9966
***->>>-----<<<-***
Epoch : 12
    Training Loss=> 0.00000180,    Training ACC 1.0000
Validation Loss=> 0.00006476, Validation ACC 0.9967
***->>>-----<<<-***
Epoch : 13
    Training Loss=> 0.00000154,    Training ACC 1.0000
Validation Loss=> 0.00006373, Validation ACC 0.9967
***->>>-----<<<-***
Epoch : 14
    Training Loss=> 0.00000133,    Training ACC 1.0000
Validation Loss=> 0.00007169, Validation ACC 0.9967
***->>>-----<<<-***
Epoch : 15
    Training Loss=> 0.00000116,    Training ACC 1.0000
Validation Loss=> 0.00006304, Validation ACC 0.9971
***->>>-----<<<-***

```

شکل 55: اجرای 5 ایپاک آخر Tanh

در ادامه نمودار مقایسه دقت این مدل‌ها برای 15 ایپاک را برای داده‌های Validation می‌آورم.

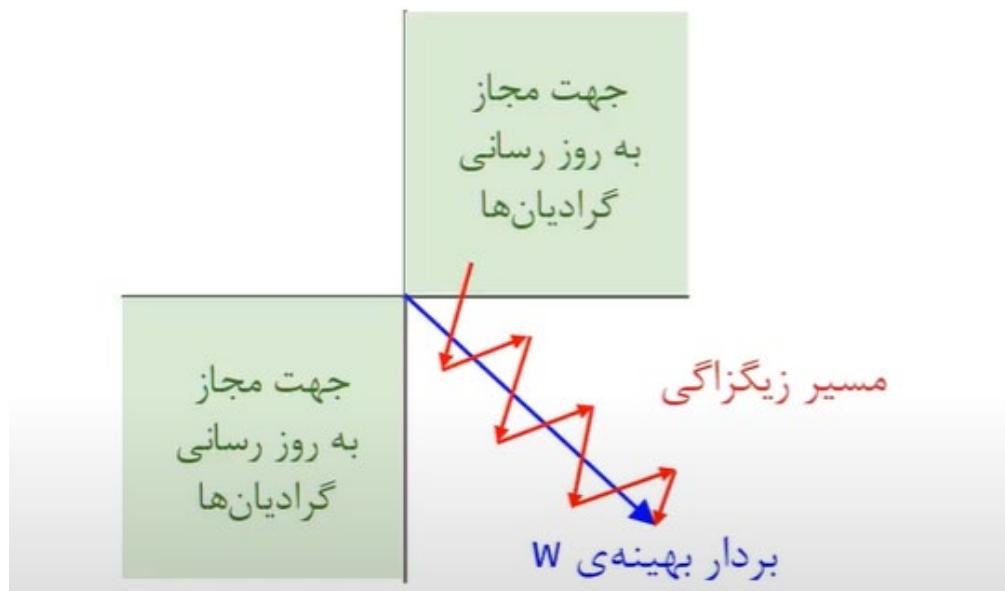


شکل ۵۶: مقایسه عملکرد سه آزمایش شده Activation Function و Sigmoid .ReLU و Tanh بر روی داده‌های Validation

➤ **نتیجه‌گیری:** همان‌طور که در شکل بالا مشخص است بهترین عملکرد را با اختلاف کمی Tanh داشته است که توانسته است خیلی سریع به حداقل دقت برسد و آن را حفظ کند و سپس با اختلاف کمی از آن ReLU توانسته است عمل کند و فقط در ابتدای کار Tanh بهتر بوده و پس از آن تقریباً مشابه هم بوده‌اند اما پس از این دو Sigmoid تقریباً با بیشترین اختلاف توانسته است تقریباً در اواخر مراحل آموزش به دقتی نزدیک به دو مدل دیگر برسد.

➤ علت این که Sigmoid از همه بدتر عمل کرده‌است به ساختار آن باز می‌گردد. چون که این تابع همان‌طور که قسمت سوال‌های عملی به آن اشاره کردم، دارای میانگین صفر برخلاف Tanh نیست (شکل این دو تابع بسیار به هم مشابه است) و به این علت باعث می‌شود که نرون‌ها با عبور از این تابع همگی مقدار مثبت پیدا کنند و حال در لایه بعدی که می‌رویم و می‌خواهیم که مشتق را حساب کنیم چون که x ها همگی مثبت هستند بنابراین با عمل backpropagation و اعمال گرادیان همه مقادیر یا مثبت می‌شوند یا منفی و این موضوع باعث می‌شود که من خیلی دیر بتوانم که به مقصد برسم و تعداد epoch‌ها بسیار زیاد شود و زمان بر شود و همچنین به دلیل هزینه‌بر بودن محاسبه خود تابع سیگموید باز هم هزینه ما را افزایش

می‌دهد. در حالی که اگر میانگین صفر بود و به مرکز صفر بود این تابع مانند Tanh می‌توانست که خیلی سریع به نتیجه برسد. در شکل زیر می‌توانید این موضوع را مشاهده کنید که باعث طولانی‌تر شدن مسیر ما می‌شود:

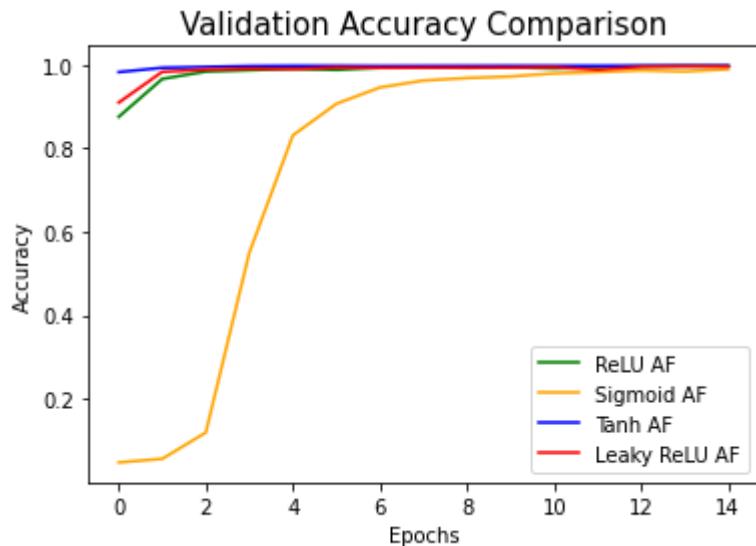


شکل 57: مشکل نداشتن میانگین صفر در Sigmoid

► حال به سراغ مقایسه بین Tanh و ReLU می‌رویم. همان‌طور که پیشتر اشاره کردم عملکرد بهتری در اینجا داشته است و علت آن هم این است که Tanh با این که مشکل اشباع شدن گرادیان را برای مقادیر خیلی کم و خیلی زیاد دارد اما در این تعداد ایپاک معمولاً این مشکل پیش نمی‌آید. اما چیزی که باعث شده بهتر از ReLU کمی بهتر عمل کند این موضوع هست که تابع ReLU در قسمت منفی‌ها عملاً به دلیل نداشتن مقدار صفر آموزش گرادیان آن نیز صفر می‌شود و در این نواحی نمی‌تواند که آموزش ببیند و همچنین در اینجا نیز ما با مقدار بسیار کمتری مشکل Zero Centered بودن یا نداشتن میانگین صفر را داریم البته می‌توانیم که از جایگزین‌های آن مانند Leaky ReLU استفاده کنیم که کمی بهتر عمل کند. (من نتایج آن را هم در ادامه قرار می‌دهم)

► اما موردی که بسیار مهم است و باعث می‌شود که ما معمولاً ReLU را انتخاب کنیم سریع‌تر کردن بسیار زیاد محاسبات است و زمان محاسبه که تا 6 برابر نیز می‌تواند سریع‌تر باشد. (آزمایش‌ها در شبکه‌های مختلف نشان داده است)

در ادامه می‌توانید مقایسه تمامی حالات بالا را با Leaky ReLU مشاهده کنید:



شکل ۵۸: مقایسه تمامی حالت خواسته شده در سوال با Leaky ReLU

توجه: کدهای مربوط به این قسمت در فایل Practical_Part_D.ipynb قرار دارد که در پوشیده مربوط به Practical Part هست.

❖ قسمت E :

در این قسمت از سوال از ما خواسته شده است که عملکرد شبکه و مدل طراحی شده را در دو حالت استفاده از Optimizer های Adam و Gradient Descent با هم مقایسه کنیم.
برای این کار من از همان تابع استفاده شده در قسمت قبل یعنی model_training() که خودم ساختم استفاده کردم و در قسمت قبل قرار دادم و اکنون من مجدد دیگر قرار نمی‌دهم. سپس قسمت مربوط به اجرای آن را به صورت زیر تغییر دادم:

```

1. from torch.optim.lr_scheduler import MultiStepLR
2.
3.
4. learning_rate = 0.00073
5. epochs = 15
6.
7. criterion = nn.CrossEntropyLoss()
8.
9. # scheduler_relu = MultiStepLR(optimizer_relu, milestones=[30, 70], gamma=0.1)
10. #####
11. model_sgd = CnnModelReLU().to(CUDA)
12. optimizer_sgd = torch.optim.SGD(model_sgd.parameters(), lr=0.073)
13.

```

```

14. print("### *** SGD Optimizer : ===>> *** ###")
15. train_loss_sigmoid, val_loss_sigmoid, train_acc_sigmoid, val_acc_sigmoid = model_
    training(model=model_sgd, epochs=epochs, optimizer=optimizer_sgd, scheduler=None,
    criterion=criterion)
16. #####
17.
18. model_adam = CnnModelReLU().to(CUDA)
19. optimizer_adam = torch.optim.Adam(model_adam.parameters(), lr=learning_rate)
20. print("### *** Adam Optimizer : ===>> *** ###")
21. train_loss_relu, val_loss_relu, train_acc_relu, val_acc_relu = model_training(mod
    el=model_adam, epochs=epochs, optimizer=optimizer_adam, scheduler=None, criterion
    =criterion)
22. #####

```

من ابتدا سعی کردم که با همان مقدار SGD ای که برای Learning Rate ای که برای Adam قرار داده بودم برای هم استفاده کنم. جواب نداد و پس از 15 ایپاک به دقیقی حدود 10 درصد رسید که مناسب نبود برای همین با چندین آزمایش دیدم که $LR=0.073$ دقت نسبتاً خوبی را می‌دهد و من از این مقدار استفاده کردم.

پس از اجرا می‌توانید نتایج تعدادی از ایپاک‌ها را در زیر مشاهده کنید:

```

### *** SGD Optimizer : ===>> *** ###
Epoch : 1
    Training Loss=> 0.01134958,      Training ACC 0.0983
    Validation Loss=> 0.01059354,   Validation ACC 0.2297
***->>-----<<<-***

Epoch : 2
    Training Loss=> 0.00813178,      Training ACC 0.3115
    Validation Loss=> 0.00658172,   Validation ACC 0.4240
***->>-----<<<-***

Epoch : 3
    Training Loss=> 0.00487016,      Training ACC 0.5644
    Validation Loss=> 0.00425241,   Validation ACC 0.6180
***->>-----<<<-***

Epoch : 4
    Training Loss=> 0.00241851,      Training ACC 0.7712
    Validation Loss=> 0.00160299,   Validation ACC 0.8541
***->>-----<<<-***

Epoch : 5
    Training Loss=> 0.00146818,      Training ACC 0.8649
    Validation Loss=> 0.00098863,   Validation ACC 0.9245
***->>-----<<<-***

```

شکل 59: پنج ایپاک اول SGD

```
### *** Adam Optimizer : ==>> *** ###
Epoch : 1
    Training Loss=> 0.00446563,      Training ACC 0.6303
Validation Loss=> 0.00085632, Validation ACC 0.9317
***->>>-----<<<-***

Epoch : 2
    Training Loss=> 0.00043400,      Training ACC 0.9662
Validation Loss=> 0.00036000, Validation ACC 0.9718
***->>>-----<<<-***

Epoch : 3
    Training Loss=> 0.00016333,      Training ACC 0.9880
Validation Loss=> 0.00021407, Validation ACC 0.9855
***->>>-----<<<-***

Epoch : 4
    Training Loss=> 0.00008866,      Training ACC 0.9938
Validation Loss=> 0.00014057, Validation ACC 0.9908
***->>>-----<<<-***

Epoch : 5
    Training Loss=> 0.00004882,      Training ACC 0.9965
Validation Loss=> 0.00013949, Validation ACC 0.9916
***->>>-----<<<-***
```

شکل 60: پنج اپیاک اول Adam

```
Epoch : 10
    Training Loss=> 0.00016562,      Training ACC 0.9893
Validation Loss=> 0.00028547,  Validation ACC 0.9825
***->>>-----<<<-***

Epoch : 11
    Training Loss=> 0.00018487,      Training ACC 0.9899
Validation Loss=> 0.00028706,  Validation ACC 0.9825
***->>>-----<<<-***

Epoch : 12
    Training Loss=> 0.00010381,      Training ACC 0.9934
Validation Loss=> 0.00029281,  Validation ACC 0.9805
***->>>-----<<<-***

Epoch : 13
    Training Loss=> 0.00007010,      Training ACC 0.9966
Validation Loss=> 0.00022567,  Validation ACC 0.9890
***->>>-----<<<-***

Epoch : 14
    Training Loss=> 0.00005380,      Training ACC 0.9973
Validation Loss=> 0.00026501,  Validation ACC 0.9846
***->>>-----<<<-***

Epoch : 15
    Training Loss=> 0.00004318,      Training ACC 0.9980
Validation Loss=> 0.00022692,  Validation ACC 0.9889
***->>>-----<<<-***
```

شکل 61: پنج اپیاک آخر SGD

```

***->>-----<<<-***  

Epoch : 10  

    Training Loss=> 0.00005181,      Training ACC 0.9956  

Validation Loss=> 0.00023363, Validation ACC 0.9861  

***->>-----<<<-***  

Epoch : 11  

    Training Loss=> 0.00002696,      Training ACC 0.9976  

Validation Loss=> 0.00016148, Validation ACC 0.9931  

***->>-----<<<-***  

Epoch : 12  

    Training Loss=> 0.00002852,      Training ACC 0.9977  

Validation Loss=> 0.00014461, Validation ACC 0.9943  

***->>-----<<<-***  

Epoch : 13  

    Training Loss=> 0.00001189,      Training ACC 0.9992  

Validation Loss=> 0.00015974, Validation ACC 0.9920  

***->>-----<<<-***  

Epoch : 14  

    Training Loss=> 0.00000286,      Training ACC 0.9999  

Validation Loss=> 0.00014474, Validation ACC 0.9957  

***->>-----<<<-***  

Epoch : 15  

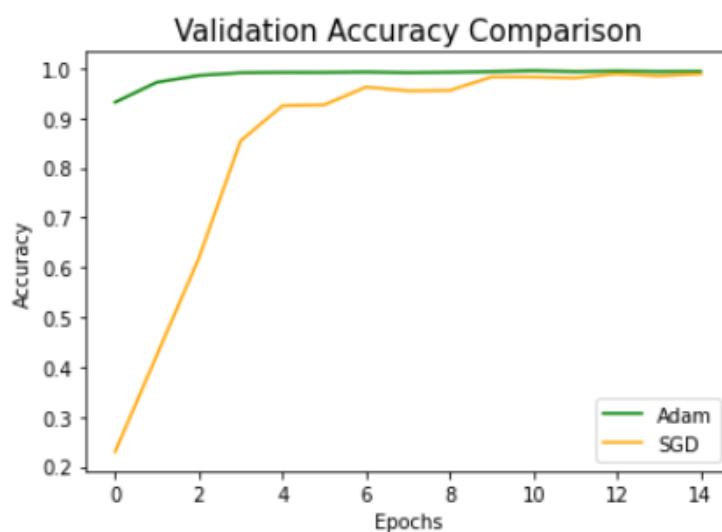
    Training Loss=> 0.00000038,      Training ACC 1.0000  

Validation Loss=> 0.00017960, Validation ACC 0.9957  

***->>-----<<<-***
```

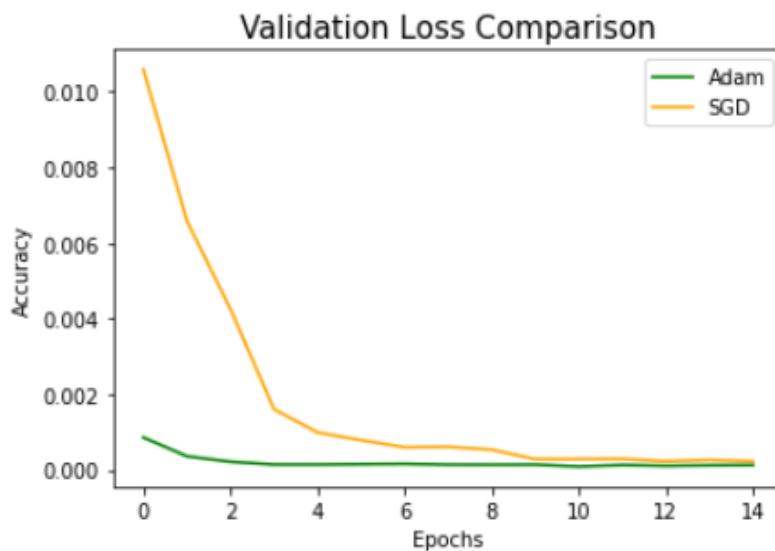
شکل 62: پنج ایپاک آخر Adam

پس از اجرای آن به نموداری مطابق شکل زیر برای داده‌های Validation می‌رسیم:



شکل 63: مقایسه دقت روی داده‌های Validation در استفاده از دوتابع Adam و SGD

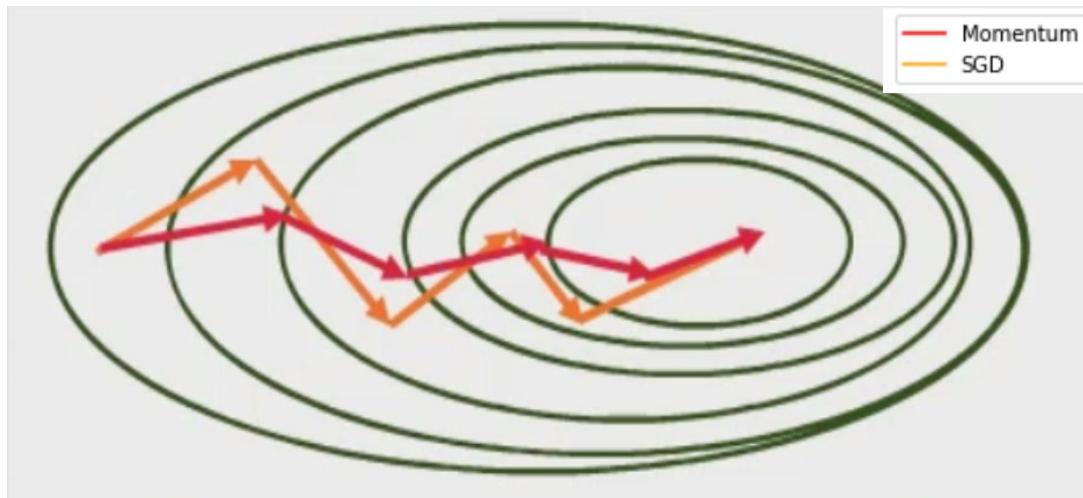
همچنین در ادامه می‌توانید نمودار Loss مربوط به این سوال را مشاهده کنید:



شکل 64: نمودار Loss مقایسه میان Adam و SGD

➤ **نتیجه‌گیری:** همان‌طور که در دو شکل بالا می‌توانید مشاهده کنید تفاوت میان دو حالت Adam و SGD در ایپاک‌های اول بسیار زیاد است و کم کم SGD این فاصله را کم می‌کند تا در نزدیکی‌های ایپاک آخر به هم می‌رسند. این موضوع نشان می‌دهد که روش SGD که خوب تقریباً می‌توان گفت اولین تابع بهینه‌سازی است که در سال 1950 میلادی معرفی شده است کمی کند عمل می‌کند تا به هدف برسد. اما روش Adam که یکی از جدیدترین روش‌های معرفی شده در سال 2014 است از نظر سرعت بسیار بهبود پیدا کرده است.

➤ اگر بخواهیم کمی دقیق‌تر به علت وجود این تفاوت میان این دو روش نگاه کنیم علت به این موضوع باز می‌گردد که روش Adam می‌توان گفت که ترکیبی از دو روش RMSProp و روش Momentum است. بخواهیم نگاه کوچکی به این دو روش داشته باشیم می‌توانم بگوییم که ابتدا در ممنتوں کاری که انجام دادند این بود که به نوعی حافظه را به تابع اضافه کردند و تابع یک تاریخچه‌ای از مسیری که طی کرده بود را در اختیار داشت و به این صورت می‌توانست مسیری که طی کرده است را نگاه کند و عملکرد خودش را با آن تنظیم کند و این موضوع باعث شد تا کمی عملکرد آن بهبود پیدا کند نسبت به SGD معمولی همان‌طور که در شکل زیر می‌توانید مقایسه‌ای بین این دو روش را مشاهده نمایید:



شکل 65: مقایسه میان عملکرد دو روش Momentum و SGD

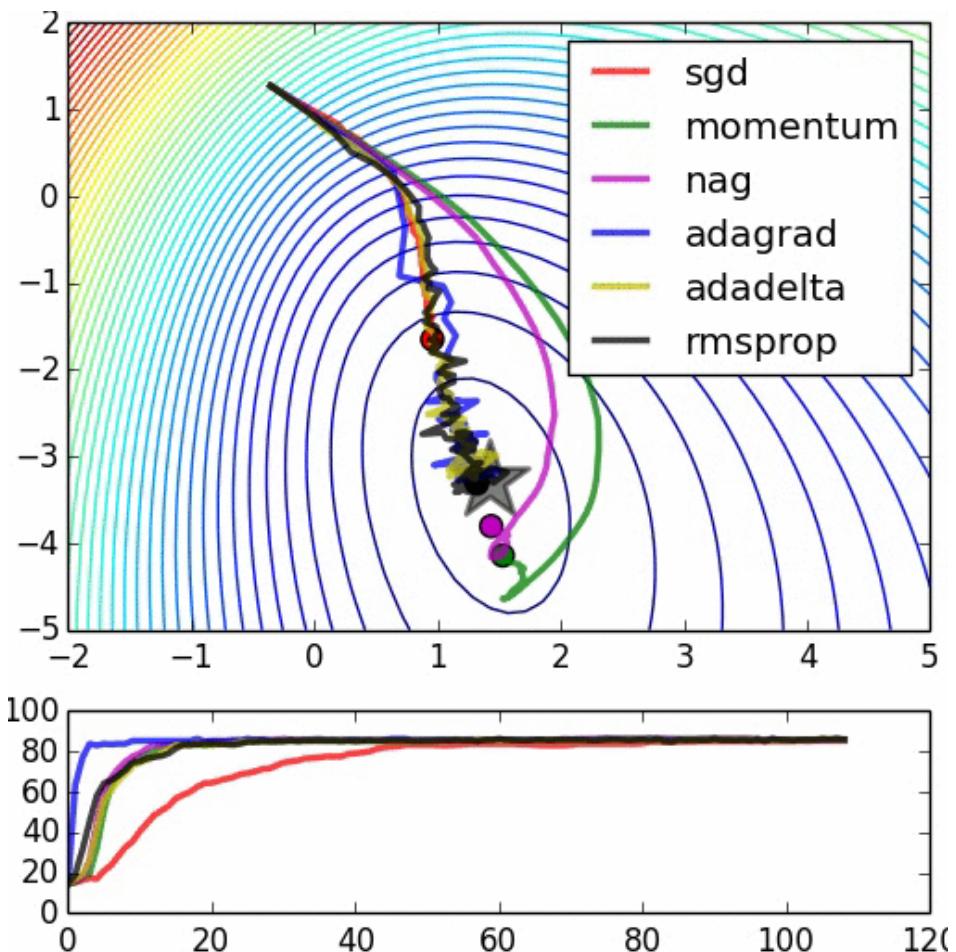
همان طور که در شکل بالا می‌توانید مشاهده کنید روش Momentum راه راست‌تری را دنبال می‌کند و به این شکل سریع‌تر به پاسخ می‌رسد و روش SGD در مینیمم‌های محلی و... گیر می‌کند و کمی طول می‌کشد تا به جواب برسد.

پس از این روش، روش RMSProp تغییری که داد این بود که آمد و نسبت به روش پیشین خود که Adagrad بود بهبودهایی را داد به این صورت که در AdaGrad مرحله به مرحله میزان Learning Rate کم می‌شود و این باعث می‌شود که تا در مراحل آخر انقدر میزان LR کم شود که عملاً دیگر یادگیری روی نمی‌دهد حال در RMSProp این موضوع را کمی تعديل کردند که میزان LR میزان کمتری و متناسب با مقادیر قبلی مقدار کمی کاهش یابد و به این شکل عملکرد را بهبود دادند.

سپس در Adam آمدند و این دو روش را با هم ترکیب کردند و به این شکل باعث شدند که سرعت Optimizer به میزان بسیار زیادی بهبود یابد و عملکرد مناسبی داشته باشد.

در شکل زیر می‌توانید مقایسه‌ای میان عملکرد این ۳تابع SGD و AdaGrad و RMSProb را مشاهده کنید. و همان‌طور که در نمودار پایین شکل مشاهده می‌کنید SGD پایین‌ترین سرعت را در میان همگی دارد. این شکل در واقع یک شکل GIF است و من [لینک](#) نسخه آنلاین آن را به همراه چندتا شکل دیگر که برای درک عملکرد این توابع مفید است را قرار می‌دهم.¹

<http://www.denizyuret.com/2015/03/alec-radford-animations-for.html>¹



شکل 66: مقایسه میان عملکرد میان روش‌های مختلف Optimization

من سعی کردم تا به ساده‌ترین و کوتاه‌ترین شکل ممکن اتفاق پیش آمده در اینجا را توجیه کنم. و همان‌طور که نتایج به دست آمده نشان می‌دهد در کل روش Adam سرعت بیشتری دارد و سریع‌تر ما را به پاسخ می‌رساند و به علاوه ویژگی آن که باعث می‌شود Learning Rate ما به تدریج کاهش یابد نیز ویژگی بسیار مفیدی هست و تاثیر بسیار خوبی روی عملکرد این روش دارد.

همچنین در آخر من نیز روی داده‌های تست نیز آزمایش کردم هرچند که مدل به طور کامل آموزش ندیده اما نتیجه آن خالی از لطف نیست و من آن را در ادامه قرار می‌دهم:

```

#### *** Adam Optimizer : ===>> *** ####
*****
Test Loss: 0.00117285, Test ACC: 0.9424
*****
#### *** SGD Optimizer : ===>> *** ####
*****
Test Loss: 0.00207474, Test ACC: 0.9064
*****

```

شکل 67: دقتهای به دست آمده بر روی مجموعه Test

➤ **توجه:** فایلهای این قسمت از سوال در فایل Practical_Part_E_Final.ipynb موجود در پوشه آپلود شده وجود دارد.

❖ قسمت F :

➤ **سوال:** در این قسمت از ما خواسته شده است که عملکرد مدل طراحی شده را با اضافه کردن لایهای Dropout بسنجیم.

➤ **پاسخ:** برای انجام این کار من اقدام به طراحی 2 مدل اضافه‌تر علاوه بر مدل اصلی ام کرده‌ام.
✓ **نمونه اول:**

اضافه کردن یک لایه Dropout بعد از اولین لایه خطی. این مدل به نام CnnModelDrop1 هست.

کد آن را می‌توانید در زیر مشاهده نمایید:

```

35 class CnnModelDrop1(nn.Module):
36     def __init__(self):
37         super(CnnModelDrop1, self).__init__()
38         self.layer1 = nn.Sequential(
39             nn.Conv2d(in_channels=3, out_channels=73, kernel_size=5, stride=1, padding=0),
40             nn.ReLU(),
41         )
42         self.layer2 = nn.Sequential(
43             nn.Conv2d(in_channels=73, out_channels=153, kernel_size=3, stride=1, padding=0),
44             nn.ReLU(),
45             nn.MaxPool2d(kernel_size=2, stride=2),
46         )
47         self.layer3 = nn.Sequential(
48             nn.Conv2d(in_channels=153, out_channels=273, kernel_size=3, stride=1, padding=1),
49             nn.ReLU(),
50             nn.MaxPool2d(kernel_size=2, stride=2),
51         )
52         self.fc1 = nn.Linear(6 * 6 * 273, 500) # 1: Stride
53         self.relu = nn.ReLU()
54         self.drop1 = nn.Dropout(p=0.5)
55         self.fc2 = nn.Linear(500, 500) # 1: Stride
56         self.softmax = nn.LogSoftmax(dim=1)
57
58     def forward(self, x):
59         out = self.layer1(x)
60         out = self.layer2(out)
61         out = self.layer3(out)
62
63         out = torch.flatten(out, 1)
64         out = self.fc1(out)
65         out = self.relu(out)
66         out = self.drop1(out)
67         out = self.fc2(out)
68         # out = self.softmax(out)
69         return out

```

شکل 68: کد مربوط به نمونه اول با یک لایه Dropout

✓ نمونه دوم:

اضافه کردن دو لایه Dropout : یکی بعد از اولین لایه خطی + دیگری بعد از اتمام لایه‌های کانولوشن. این مدل به نام CnnModelDrop2 است.

در ادامه می‌توانید کد این نمونه را مشاهده نمایید:

```

72 class CnnModelDrop2(nn.Module):
73     def __init__(self):
74         super(CnnModelDrop2, self).__init__()
75         self.layer1 = nn.Sequential(
76             nn.Conv2d(in_channels=3, out_channels=73, kernel_size=5, stride=1, padding=0),
77             nn.ReLU(),
78         )
79         self.layer2 = nn.Sequential(
80             nn.Conv2d(in_channels=73, out_channels=153, kernel_size=3, stride=1, padding=0),
81             nn.ReLU(),
82             nn.MaxPool2d(kernel_size=2, stride=2),
83         )
84         self.layer3 = nn.Sequential(
85             nn.Conv2d(in_channels=153, out_channels=273, kernel_size=3, stride=1, padding=1),
86             nn.ReLU(),
87             nn.MaxPool2d(kernel_size=2, stride=2),
88         )
89         self.drop2 = nn.Dropout2d(p=0.5)
90         self.fc1 = nn.Linear(6 * 6 * 273, 500) # 1: Stride
91         self.relu = nn.ReLU()
92         self.drop1 = nn.Dropout(p=0.5)
93         self.fc2 = nn.Linear(500, 43) # 1: Stride
94         # self.softmax = nn.LogSoftmax(dim=1)
95
96     def forward(self, x):
97         out = self.layer1(x)
98         out = self.layer2(out)
99         out = self.layer3(out)
100        out = self.drop2(out)
101
102        out = torch.flatten(out, 1)
103        out = self.fc1(out)
104        out = self.relu(out)
105        out = self.drop1(out)
106        out = self.fc2(out)
107        # out = self.softmax(out)
108        return out

```

شکل 69: کد مربوط به نمونه دوم با دو لایه Dropout

توجه: مقدار $p=0.5$ به عنوان آرگومان ورودی لایه Dropout در واقع احتمال Drop شدن را بیان می‌کند و با مقدار 0.5 که من به آن داده‌ام یعنی نیمی از آن‌ها Drop شوند.

پس از اجرای این کدها نتایج به صورت زیر حاصل می‌شود:

```

### *** Dropout 2 : ===>> *** ##
Epoch : 1
    Training Loss=> 0.00611908,      Training ACC 0.4891
Validation Loss=> 0.00178814, Validation ACC 0.8406
***->>>-----<<<-***
Epoch : 2
    Training Loss=> 0.00098571,      Training ACC 0.9095
Validation Loss=> 0.00072243, Validation ACC 0.9422
***->>>-----<<<-***
Epoch : 3
    Training Loss=> 0.00043818,      Training ACC 0.9616
Validation Loss=> 0.00044680, Validation ACC 0.9603
***->>>-----<<<-***
Epoch : 4
    Training Loss=> 0.00026725,      Training ACC 0.9762
Validation Loss=> 0.00030177, Validation ACC 0.9764
***->>>-----<<<-***
Epoch : 5
    Training Loss=> 0.00019681,      Training ACC 0.9831
Validation Loss=> 0.00025838, Validation ACC 0.9818
***->>>-----<<<-***

```

شکل 70: پنج ایپاک اول مدل دارای 2 عدد Dropout

```

### *** Dropout 1 : ===>> *** ##
Epoch : 1
    Training Loss=> 0.00821872,      Training ACC 0.3673
Validation Loss=> 0.00265961, Validation ACC 0.7571
***->>>-----<<<-***
Epoch : 2
    Training Loss=> 0.00130927,      Training ACC 0.8811
Validation Loss=> 0.00076276, Validation ACC 0.9362
***->>>-----<<<-***
Epoch : 3
    Training Loss=> 0.00045516,      Training ACC 0.9606
Validation Loss=> 0.00040241, Validation ACC 0.9695
***->>>-----<<<-***
Epoch : 4
    Training Loss=> 0.00024593,      Training ACC 0.9794
Validation Loss=> 0.00032525, Validation ACC 0.9736
***->>>-----<<<-***
Epoch : 5
    Training Loss=> 0.00017354,      Training ACC 0.9848
Validation Loss=> 0.00022728, Validation ACC 0.9825
***->>>-----<<<-***

```

شکل 71: پنج ایپاک اول مدل دارای 1 عدد Dropout

```

### *** Normal : ==>> *** ####
Epoch : 1
    Training Loss=> 0.00642903,     Training ACC 0.5053
Validation Loss=> 0.00113797, Validation ACC 0.9055
***->>-----<<<-***

Epoch : 2
    Training Loss=> 0.00054618,     Training ACC 0.9547
Validation Loss=> 0.00035466, Validation ACC 0.9763
***->>-----<<<-***

Epoch : 3
    Training Loss=> 0.00017846,     Training ACC 0.9864
Validation Loss=> 0.00029832, Validation ACC 0.9783
***->>-----<<<-***

Epoch : 4
    Training Loss=> 0.00010077,     Training ACC 0.9926
Validation Loss=> 0.00017959, Validation ACC 0.9881
***->>-----<<<-***

Epoch : 5
    Training Loss=> 0.00005083,     Training ACC 0.9965
Validation Loss=> 0.00015413, Validation ACC 0.9917
***->>-----<<<-***
```

شکل 72: پنج ایپاک اول حالت عادی بدون Dropout

```

***->>-----<<<-***
Epoch : 95
    Training Loss=> 0.00000000,     Training ACC 1.0000
Validation Loss=> 0.00016470, Validation ACC 0.9960
***->>-----<<<-***

Epoch : 96
    Training Loss=> 0.00000000,     Training ACC 1.0000
Validation Loss=> 0.00016488, Validation ACC 0.9960
***->>-----<<<-***

Epoch : 97
    Training Loss=> 0.00000000,     Training ACC 1.0000
Validation Loss=> 0.00016782, Validation ACC 0.9960
***->>-----<<<-***

Epoch : 98
    Training Loss=> 0.00000000,     Training ACC 1.0000
Validation Loss=> 0.00016528, Validation ACC 0.9960
***->>-----<<<-***

Epoch : 99
    Training Loss=> 0.00000000,     Training ACC 1.0000
Validation Loss=> 0.00025664, Validation ACC 0.9960
***->>-----<<<-***

Epoch : 100
    Training Loss=> 0.00000000,     Training ACC 1.0000
Validation Loss=> 0.00016552, Validation ACC 0.9960
***->>-----<<<-***
```

شکل 73: پنج ایپاک آخر مدل عادی بدون Dropout

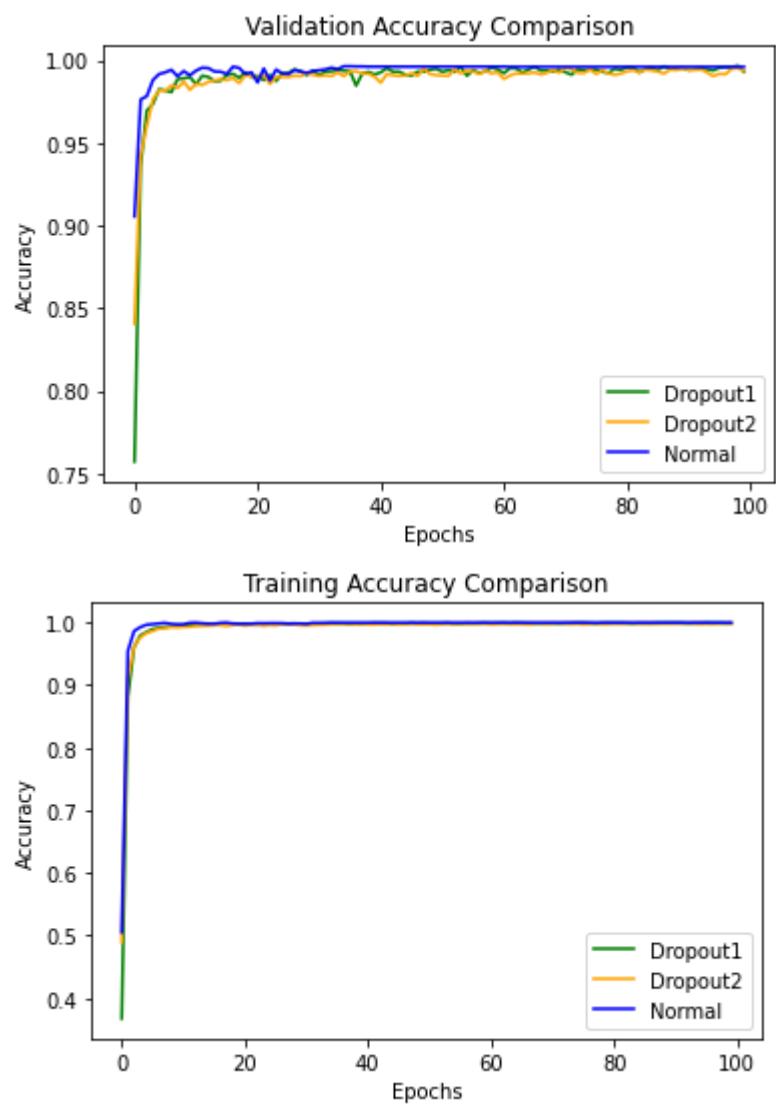
```
***->>>-----<<<-***  
Epoch : 95  
    Training Loss=> 0.00000900,    Training ACC 0.9991  
Validation Loss=> 0.00017381, Validation ACC 0.9939  
***->>>-----<<<-***  
Epoch : 96  
    Training Loss=> 0.00000473,    Training ACC 0.9996  
Validation Loss=> 0.00014363, Validation ACC 0.9955  
***->>>-----<<<-***  
Epoch : 97  
    Training Loss=> 0.00001088,    Training ACC 0.9991  
Validation Loss=> 0.00011218, Validation ACC 0.9959  
***->>>-----<<<-***  
Epoch : 98  
    Training Loss=> 0.00000544,    Training ACC 0.9994  
Validation Loss=> 0.00014511, Validation ACC 0.9958  
***->>>-----<<<-***  
Epoch : 99  
    Training Loss=> 0.00001141,    Training ACC 0.9989  
Validation Loss=> 0.00010280, Validation ACC 0.9968  
***->>>-----<<<-***  
Epoch : 100  
    Training Loss=> 0.00001132,    Training ACC 0.9989  
Validation Loss=> 0.00015608, Validation ACC 0.9931  
***->>>-----<<<-***
```

شکل 74: پنج ایپاک آخر مدل با 1 لایه Dropout

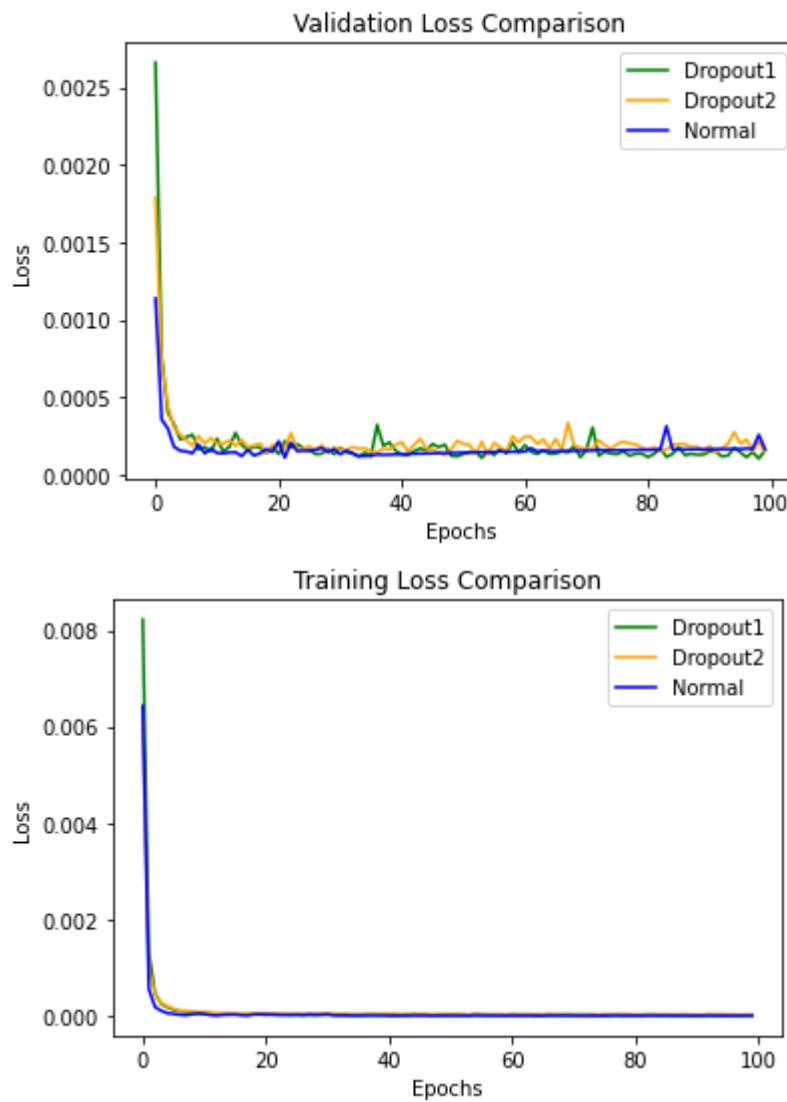
```
***->>-----<<<-***  
Epoch : 95  
    Training Loss=> 0.00003292,      Training ACC 0.9978  
Validation Loss=> 0.00027318, Validation ACC 0.9904  
***->>-----<<<-***  
Epoch : 96  
    Training Loss=> 0.00002626,      Training ACC 0.9978  
Validation Loss=> 0.00020066, Validation ACC 0.9918  
***->>-----<<<-***  
Epoch : 97  
    Training Loss=> 0.00002471,      Training ACC 0.9977  
Validation Loss=> 0.00022649, Validation ACC 0.9916  
***->>-----<<<-***  
Epoch : 98  
    Training Loss=> 0.00002113,      Training ACC 0.9982  
Validation Loss=> 0.00015035, Validation ACC 0.9949  
***->>-----<<<-***  
Epoch : 99  
    Training Loss=> 0.00001591,      Training ACC 0.9988  
Validation Loss=> 0.00021351, Validation ACC 0.9954  
***->>-----<<<-***  
Epoch : 100  
    Training Loss=> 0.00001871,      Training ACC 0.9985  
Validation Loss=> 0.00014527, Validation ACC 0.9936  
***->>-----<<<-***
```

شکل 75: پنج ایپاک آخر مدل با 2 لاشه Dropout

سپس در ادامه می‌توانید نمودار نتایج به دست آمده را مشاهده نمایید:



شکل 76: نمودار دقت بر روی دو دسته داده‌های Train و نیز Validation



شکل 77: نمودار مربوط به مقدار Loss در دو مجموعه داده Validation و Train

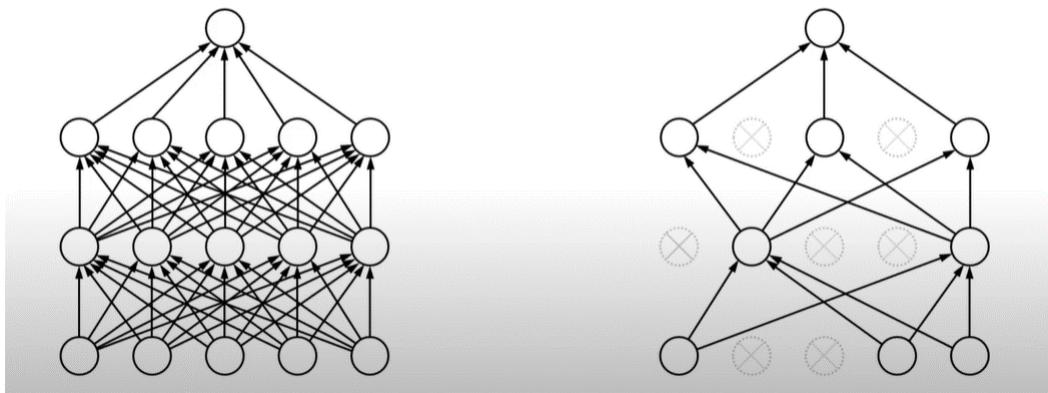
```
### *** Normal Model : ==>> *** ###
*****
Test Loss: 0.00127438, Test ACC: 0.9594
*****
### *** 1 Dropout : ==>> *** ###
*****
Test Loss: 0.00099510, Test ACC: 0.9644
*****
### *** 2 Dropout : ==>> *** ###
*****
Test Loss: 0.00135907, Test ACC: 0.9648
*****
```

شکل 78: دقت به دست آمده بر روی مجموعه داده‌های تست

► نتیجه‌گیری:

- ✓ ابتدا در مقایسه میان دو حالت یک لایه Dropout و دو لایه Dropout به این نتیجه می‌رسیم که یک لایه Dropout کمی Smooth‌تر هست نمودارش و توانسته است که مقادیر Loss کمتری را نیز به دست بیاورد. اما در دقت روی داده‌های تست هم می‌توانیم مشاهده کنیم که دقت هر دو روش Dropout مشابه هستند اما مقدار loss روش با یک لایه Dropout کمتر شده است. علت این موضوع که یک لایه Dropout کمی بهتر عمل می‌کند این است که من در قسمت سوال‌های تشریحی نیز بیان کردم که بهتر است که ما Dropout را در لایه‌های آخر شبکه به کار بگیریم و مشاهده می‌کنیم که این حالت فقط در لایه یکی مانده به آخر این کار را انجام داده است و به این شکل توانسته است بهتر عمل کند.
- ✓ سپس به مقایسه میان عملکرد Dropout و شبکه عادی می‌پردازیم که در این قسمت تفاوت بسیار کم است اما اگر به نمودار Loss و همچنین نمودار Accuracy و نیز مقادیری که در هر ایپاک به دست آمده است ما نگاه بیندازیم، مشاهده می‌کنیم که عملکرد Dropout کمی بهتر است و نمودار آن Smooth‌تر شده و این موضوع را می‌توانیم در دقت داده‌های تست موجود در شکل آخر نیز مشاهده کنیم.

یک نمای کلی از نحوه انجام Dropout را می‌توانید در شکل زیر مشاهده کنید:



شکل 79: نحوه انجام Dropout

- ✓ حال در ادامه می‌خواهم به بررسی علت این که دقت Dropout بالاتر از حالت عادی می‌شود و نیز در ایپاک‌های آخر در مجموعه valid و test مقادیر loss کمتر و نیز دقت بالاتری به دست می‌آوریم بپردازم. علت این موضوع در این است که با انجام Dropout همان‌طور که در شکل زیر مشاهده می‌کنید باعث می‌شود که شبکه ما به یک feature و ویژگی خاص وابسته نباشد و به این شکل وابستگی به یک feature خاص را از بین می‌بریم و به این شکل تعمیم و

Generalization را افزایش می‌دهیم. علت هم این است که چون هر بار ما تعدادی از نرون‌ها را صفر می‌کنیم و با این کار باعث می‌شویم که در آن مرحله آن نرون که حاوی ویژگی‌هایی هست در نظر گرفته نشود و به این شکل کار ما را بهبود می‌دهد و باعث می‌شود که با حداقل داده‌ها بتوانیم بیشترین تعداد ویژگی و Feature را استخراج کنیم. (در شکل زیر نیز می‌توانید این موضوع را برای به عنوان مثال تشخیص یک گربه مشاهده کنید.)



شکل ۸۰: در Dropout ما در هر مرحله یک سری از Feature‌ها را از بین می‌بریم و به این شکل وابستگی به یک ویژگی خاص را از بین می‌بریم.

- ✓ یک تفسیر دیگری که می‌توانیم برای عملکرد Dropout ارائه دهیم این است که این عمل باعث می‌شود که عملاً ما در هر تکرار شبکه‌های مختلفی را ایجاد کنیم یعنی در واقع در هر بار تکرار تعداد نرون‌های خاصی آموزش می‌بینند و در خروجی ما اثر دارند و این عمل مثل این است که در واقع ما تعداد زیادی شبکه را ایجاد کنیم (2^h) و سپس در آخر از میان تمامی این‌ها اکثریت بگیریم (در واقع به جای یک مدل مجموعه‌ای از مدل‌ها را آموزش می‌دهیم).
- ✓ در انتها نیز اشاره به این نکته لازم است که در اینجا چون ما overfitting نداشتیم عملکرد و تاثیر زیادی را نتوانستیم در نمودارها و دقت روی داده‌های تست مشاهده کنیم اختلاف‌ها در حد یک یا دو درصد بودند اما در حالاتی که در مدل اصلی overfitting روی دهد می‌توانیم به خوبی عملکرد استفاده از این روش را مشاهده کنیم.

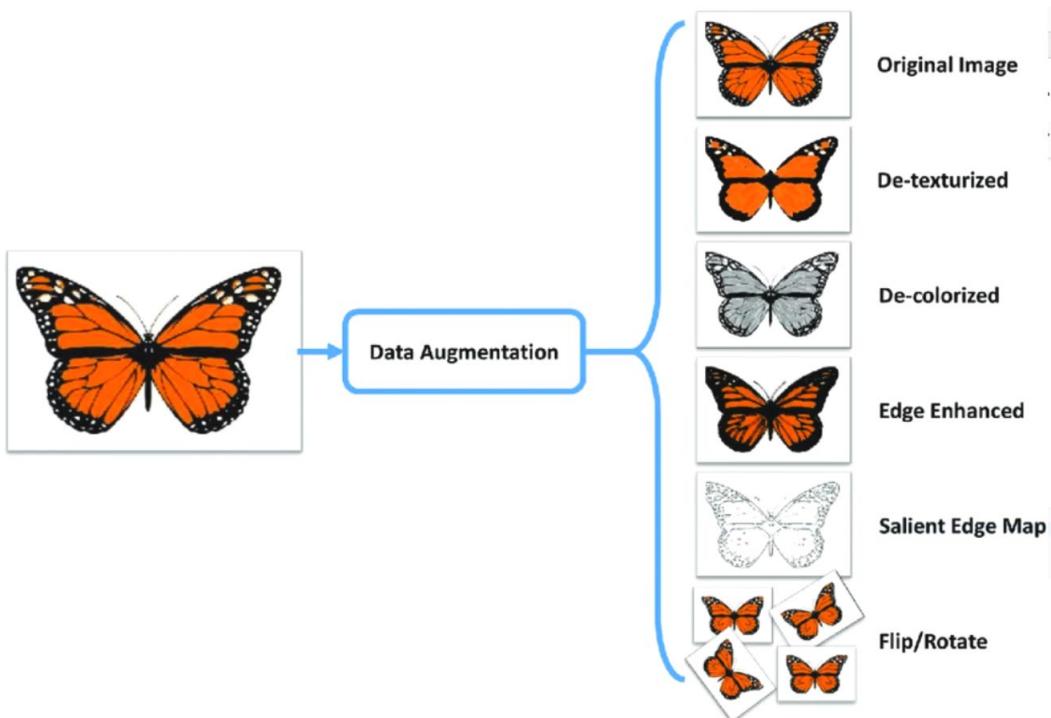
► **توجه:** کدهای این قسمت از سوال در فایل Practical_Part_F.ipynb موجود در پوشه آپلود شده می‌باشد.

❖ قسمت G ❖

در این قسمت از ما خواسته شده است که از Data Augmentation بر روی داده‌های Train و Validation استفاده کنیم و سپس نتیجه آن را مشاهده کنیم و گزارش کنیم.

پاسخ: ✓

برای انجام این کار ما از داده‌هایی که در دیتاست فعلی مان موجود داریم استفاده می‌کنیم و با تغییراتی در آن‌ها، تصاویر جدیدی را ایجاد می‌کنیم. این اعمال را می‌توانید در شکل زیر مشاهده کنید:



شکل 81: تعدادی از کارهایی که برای Augment کردن دیتاهایمان می‌توانیم انجام دهیم.

به این تصاویر جدیدی که از این طریق ایجاد می‌شوند Augmented Images می‌گویند. و این تصاویر جدید به ما امکان می‌دهند که دیتاستمان را با اضافه کردن داده‌های جدید Augment کنیم.

علت این که این روش موثر است و خوب عمل می‌کند این است که اجزا می‌دهد مدل طراحی شده ما به دیتاستی که داریم از Perspective های مختلفی نگاه کند و بنابراین می‌تواند با دقت بیشتری ویژگی‌های مرتبط را استخراج کند و همچنین ویژگی‌های بیشتری را نیز استخراج کند.

همان طور که در قسمت سوال‌های تشریحی توضیحات کامل دادم استفاده از این روش باعث می‌شود که مدل طراحی شده ما بهتر آموزش ببیند و بتواند در محیط واقعی Generalization بهتری را از خود نشان دهد و همچنین در صورت به وجود آمدن Overfitting این کار یکی از روش‌های خوب برای رفع آن است.

➤ **توجه:** با جستجو و تحقیق‌های که داشتم مطمئن هستم که عمل Data Augmentation تنها باید بر روی مجموعه Train اعمال شود. به این علت که فلسفه این است که ما مدلمان را با استفاده از داده‌های بیشتری که بر اثر چرخش و کم و زیاد کردن نور و سایر موارد تولید می‌کنیم با یک دیتا است متتنوع‌تری از داده‌های آموزشی، آموزش دهیم و به این شکل بتوانیم یک مدل بهتر و با بالاتری داشته باشیم.

در ادامه می‌توانید کدهای مربوط به اعمال Data Augmentation را بر روی مجموعه داده‌های Train مشاهده نمایید:

```
42 class ApplyTransform(Dataset):
43     """
44     Apply Transformations to a Dataset
45
46     Argumants:
47
48     """
49     def __init__(self, dataset, transform=None, target_transform=None):
50         self.dataset = dataset
51         self.transform = transform
52         self.target_transform = target_transform
53
54     def __getitem__(self, idx):
55         sample, target = self.dataset[idx]
56         if self.transform is not None:
57             sample = self.transform(sample)
58         if self.target_transform is not None:
59             target = self.target_transform(target)
60         return sample, target
61
62     def __len__(self):
63         return len(self.dataset)
```

شکل 82: مدلی که برای اعمال Transform‌های مختلف بر روی مجموعه Train طراحی کردہ‌ام

سپس از کلاس بالا برای اعمال Data Augmentation به صورت زیر بر روی داده‌های Train استفاده می‌کنم:

```

15 # For Data Augmentation
16 train_transforms = transforms.Compose([
17     transforms.ToPILImage(),
18     transforms.Resize((image_size, image_size)),
19     transforms.RandomHorizontalFlip(),
20     transforms.RandomRotation(10),
21     transforms.RandomAffine(0, shear=10, scale=(0.8, 1.2)),
22     transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
23     transforms.ToTensor(),
24     transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
25     # transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
26 ])
27
28 train_dataset = ImageFolderNew(root=train_dir)
29
30
31 number_of_train = (0.8 * len(train_dataset)).__int__()
32 number_of_valid = (0.2 * len(train_dataset)).__int__() + 1
33
34 data_train, data_valid = random_split(train_dataset, [number_of_train, number_of_valid])
35
36 train_dataset = ApplyTransform(data_train, transform=train_transforms)
37
38 train_loader_aug = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)

```

شکل 83: اعمال Transformation‌های مختلف برای ایجاد Data Augmentation بر روی داده‌های آموزشی

در زیر کاربرد تعدادی از Transformation‌های استفاده شده برای Data Augmentation را قرار می‌دهم:

| نام Transform | عمل انجام شده |
|-----------------------------|---|
| RandomHorizontalFlip | به صورت تصادفی تعدادی از تصاویر را به صورت افقی flip می‌کند. |
| RandomRotation | به صورت تصادفی تعدادی از تصاویر را به میزان مقدار مشخص شده (10 درجه) می‌چرخاند. |
| RandomAffine | به صورت تصادفی تعدادی از تصاویر را با میزان‌های مشخص شده نوردهی‌شان را کم یا زیاد می‌کند. |
| ColorJitter | کنتراست تصاویر را زیاد می‌کند با مقدارهای داده شده به آن |

به عنوان مثال تعدادی از داده‌های دیتابست را قبل از انجام این موارد در زیر می‌توانید ببینید:



شکل ۸۴: دیتاست آموزش بدون Data Augmentation

پس از اعمال موارد گفته شده دیتاهای به شکل زیر تبدیل می‌شوند:



شکل ۸۵: دیتاست آموزش بعد از اعمال Data Augmentation

همان‌طور که در شکل‌های بالا می‌توانید مشاهده کنید در حالت اعمال Data Augmentation تصاویر چرخش کمی با همان مقدار ۱۰ درجه گفته شده داشته‌اند همچنین تغییر کنتراست و رنگ تصاویر به چشم می‌آید.

لازم نیست که در این سوال تغییری در مدل‌مان اعمال کنیم اما من این سوال را در دو حالت مختلف حل می‌کنم.

► حالت اول:

اعمال Data Augmentation تنها و بر روی مدل اولیه‌ای که طراحی کردیم.

در زیر می‌توانید مدل اولیه را ببینید:

```

1  class CnnModelNormal(nn.Module):
2      def __init__(self):
3          super(CnnModelNormal, self).__init__()
4          self.layer1 = nn.Sequential(
5              nn.Conv2d(in_channels=3, out_channels=73, kernel_size=5, stride=1, padding=0),
6              nn.ReLU(),
7          )
8          self.layer2 = nn.Sequential(
9              nn.Conv2d(in_channels=73, out_channels=153, kernel_size=3, stride=1, padding=0),
10             nn.ReLU(),
11             nn.MaxPool2d(kernel_size=2, stride=2),
12         )
13         self.layer3 = nn.Sequential(
14             nn.Conv2d(in_channels=153, out_channels=273, kernel_size=3, stride=1, padding=1),
15             nn.ReLU(),
16             nn.MaxPool2d(kernel_size=2, stride=2),
17         )
18         self.fc1 = nn.Linear(6 * 6 * 273, 500) # 1: Stride
19         self.relu = nn.ReLU()
20         self.fc2 = nn.Linear(500, 43) # 1: Stride
21         self.softmax = nn.LogSoftmax(dim=1)
22
23     def forward(self, x):
24         out = self.layer1(x)
25         out = self.layer2(out)
26         out = self.layer3(out)
27
28         out = torch.flatten(out, 1)
29         out = self.fc1(out)
30         out = self.relu(out)
31         out = self.fc2(out)
32         # out = self.softmax(out)
33         return out

```

شکل 86: مدل اولیه استفاده شده برای Data Augmentation و بدون

➤ حالت دوم:

اعمال Dropout + Data Augmentation بر روی مدل اولیه

در زیر می‌توانید کد مربوطه را مشاهده کنید:

```

35  class CnnModelDrop1(nn.Module):
36      def __init__(self):
37          super(CnnModelDrop1, self).__init__()
38          self.layer1 = nn.Sequential(
39              nn.Conv2d(in_channels=3, out_channels=73, kernel_size=5, stride=1, padding=0),
40              nn.ReLU(),
41          )
42          self.layer2 = nn.Sequential(
43              nn.Conv2d(in_channels=73, out_channels=153, kernel_size=3, stride=1, padding=0),
44              nn.ReLU(),
45              nn.MaxPool2d(kernel_size=2, stride=2),
46          )
47          self.layer3 = nn.Sequential(
48              nn.Conv2d(in_channels=153, out_channels=273, kernel_size=3, stride=1, padding=1),
49              nn.ReLU(),
50              nn.MaxPool2d(kernel_size=2, stride=2),
51          )
52          self.fc1 = nn.Linear(6 * 6 * 273, 500) # 1: Stride
53          self.relu = nn.ReLU()
54          self.drop1 = nn.Dropout(p=0.5)
55          self.fc2 = nn.Linear(500, 43) # 1: Stride
56          self.softmax = nn.LogSoftmax(dim=1)
57
58      def forward(self, x):
59          out = self.layer1(x)
60          out = self.layer2(out)
61          out = self.layer3(out)
62
63          out = torch.flatten(out, 1)
64          out = self.fc1(out)
65          out = self.relu(out)
66          out = self.drop1(out)
67          out = self.fc2(out)
68          # out = self.softmax(out)
69          return out

```

شکل 87: کد مربوط به مدل دارای Dropout برای استفاده با Data Augmentation

سپس با استفاده از دستورات زیر و تابع model_training که پیشتر معرفی کردم اقدام به آموزش این حالات مختلف می‌کنیم:

```

1. from torch.optim.lr_scheduler import MultiStepLR
2.
3.
4. learning_rate = 0.00073
5. epochs = 100
6.
7. criterion = nn.CrossEntropyLoss()
8.
9. # scheduler_relu = MultiStepLR(optimizer_relu, milestones=[30, 70], gamma=0.1)
10. #####
11. #####
12. #####
13. model_drop_aug = CnnModelDrop1().to(CUDA)
14. optimizer_drop_aug = torch.optim.Adam(model_drop_aug.parameters(), lr=learning_rate)
15. print("### *** Dropout 1 + Augmentation : ==>> *** ###")
16. train_loss_drop_aug, val_loss_drop_aug, train_acc_drop_aug, val_acc_drop_aug = model_training(model=model_drop_aug, epochs=epochs, optimizer=optimizer_drop_aug, scheduler=None, criterion=criterion, train_loader=train_loader_aug)
17. #####

```

```

18. model_aug = CnnModelNormal().to(CUDA)
19. optimizer_aug = torch.optim.Adam(model_aug.parameters(), lr=learning_rate)
20. print("### *** Normal + Augmentation : ===>> *** ###")
21. train_loss_aug, val_loss_aug, train_acc_aug, val_acc_aug = model_training(model=model_aug, epochs=epochs, optimizer=optimizer_aug, scheduler=None, criterion=criterion, train_loader=train_loader_aug)
22. #####
23. model_normal = CnnModelNormal().to(CUDA)
24. optimizer_normal = torch.optim.Adam(model_normal.parameters(), lr=learning_rate)

25. print("### *** Normal : ===>> *** ###")
26. train_loss_normal, val_loss_normal, train_acc_normal, val_acc_normal = model_training(model=model_normal, epochs=epochs, optimizer=optimizer_normal, scheduler=None, criterion=criterion, train_loader=train_loader)
27. #####

```

پس از اجرای کد فوق نتایج به صورت زیر به دست می‌آید:

```

### *** Dropout 1 + Augmentation : ===>> *** ###
Epoch : 1
Training Loss=> 0.00727985, Training ACC 0.3836
Validation Loss=> 0.00337948, Validation ACC 0.6790
***->>>-----<<<-***

Epoch : 2
Training Loss=> 0.00278192, Training ACC 0.7279
Validation Loss=> 0.00154843, Validation ACC 0.8497
***->>>-----<<<-***

Epoch : 3
Training Loss=> 0.00158518, Training ACC 0.8430
Validation Loss=> 0.00091860, Validation ACC 0.9079
***->>>-----<<<-***

Epoch : 4
Training Loss=> 0.00111296, Training ACC 0.8909
Validation Loss=> 0.00059470, Validation ACC 0.9487
***->>>-----<<<-***

Epoch : 5
Training Loss=> 0.00084139, Training ACC 0.9169
Validation Loss=> 0.00045507, Validation ACC 0.9614
***->>>-----<<<-***

```

شکل 88: نتیجه 5 اجرای اول Dropout + Data Augmentation

```

***->>-----<<<-***  

Epoch : 95  

    Training Loss=> 0.00007041,      Training ACC 0.9941  

Validation Loss=> 0.00003867, Validation ACC 0.9964  

***->>-----<<<-***  

Epoch : 96  

    Training Loss=> 0.00005997,      Training ACC 0.9944  

Validation Loss=> 0.00004482, Validation ACC 0.9962  

***->>-----<<<-***  

Epoch : 97  

    Training Loss=> 0.00006894,      Training ACC 0.9937  

Validation Loss=> 0.00004673, Validation ACC 0.9972  

***->>-----<<<-***  

Epoch : 98  

    Training Loss=> 0.00005592,      Training ACC 0.9949  

Validation Loss=> 0.00004480, Validation ACC 0.9968  

***->>-----<<<-***  

Epoch : 99  

    Training Loss=> 0.00005650,      Training ACC 0.9948  

Validation Loss=> 0.00005932, Validation ACC 0.9955  

***->>-----<<<-***  

Epoch : 100  

    Training Loss=> 0.00006339,      Training ACC 0.9936  

Validation Loss=> 0.00005176, Validation ACC 0.9966  

***->>-----<<<-***
```

شکل 89: نتیجه ۵ اجرای آخر Dropout + Data Augmentation

```

*****  

--->> 4666.967856168747 seconds <<<---  

*****
```

شکل 90: زمان اجرای حالت Dropout + Data Augmentation

```

*****
### *** Normal + Augmentation : ===>> *** ##
Epoch : 1
    Training Loss=> 0.00688276,    Training ACC 0.4203
Validation Loss=> 0.00237676, Validation ACC 0.7651
***->>-----<<<-***

Epoch : 2
    Training Loss=> 0.00189846,    Training ACC 0.8203
Validation Loss=> 0.00090617, Validation ACC 0.9169
***->>-----<<<-***

Epoch : 3
    Training Loss=> 0.00099979,    Training ACC 0.9067
Validation Loss=> 0.00047780, Validation ACC 0.9578
***->>-----<<<-***

Epoch : 4
    Training Loss=> 0.00069003,    Training ACC 0.9368
Validation Loss=> 0.00030635, Validation ACC 0.9769
***->>-----<<<-***

Epoch : 5
    Training Loss=> 0.00053157,    Training ACC 0.9513
Validation Loss=> 0.00025870, Validation ACC 0.9782
***->>-----<<<-***
```

شکل 91: نتیجه ۵ اجرای اول Data Augmentation تنها بر روی مدل عادی

```

***->>-----<<<-***
Epoch : 95
    Training Loss=> 0.00002405,    Training ACC 0.9981
Validation Loss=> 0.00002669, Validation ACC 0.9981
***->>-----<<<-***

Epoch : 96
    Training Loss=> 0.00002789,    Training ACC 0.9975
Validation Loss=> 0.00002773, Validation ACC 0.9981
***->>-----<<<-***

Epoch : 97
    Training Loss=> 0.00002846,    Training ACC 0.9975
Validation Loss=> 0.00002551, Validation ACC 0.9986
***->>-----<<<-***

Epoch : 98
    Training Loss=> 0.00002799,    Training ACC 0.9974
Validation Loss=> 0.00002214, Validation ACC 0.9985
***->>-----<<<-***

Epoch : 99
    Training Loss=> 0.00004424,    Training ACC 0.9960
Validation Loss=> 0.00002422, Validation ACC 0.9991
***->>-----<<<-***

Epoch : 100
    Training Loss=> 0.00004562,    Training ACC 0.9961
Validation Loss=> 0.00001847, Validation ACC 0.9986
***->>-----<<<-***
```

شکل 92: نتیجه ۵ اجرای آخر Data Augmentation تنها بر روی مدل عادی

```
*****
--->>> 4691.942150354385 seconds <<<---
*****
```

شکل 93: زمان اجرای 100 ایپاک حالت Data Augmentation بر روی مدل عادی

```
### *** Normal : ===>> *** ###
Epoch : 1
    Training Loss:=> 0.00446596,    Training ACC 0.6325
Validation Loss:=> 0.00082522, Validation ACC 0.9325
***->>>-----<<<-***
Epoch : 2
    Training Loss:=> 0.00046813,    Training ACC 0.9624
Validation Loss:=> 0.00032113, Validation ACC 0.9750
***->>>-----<<<-***
Epoch : 3
    Training Loss:=> 0.00017698,    Training ACC 0.9869
Validation Loss:=> 0.00021409, Validation ACC 0.9848
***->>>-----<<<-***
Epoch : 4
    Training Loss:=> 0.00009580,    Training ACC 0.9929
Validation Loss:=> 0.00015009, Validation ACC 0.9901
***->>>-----<<<-***
Epoch : 5
    Training Loss:=> 0.00005285,    Training ACC 0.9963
Validation Loss:=> 0.00013869, Validation ACC 0.9920
***->>>-----<<<-***
```

شکل 94: نتیجه 5 اجرای اول حالت بدون Data Augmentation

```

***->>-----<<<-***  

Epoch : 95  

    Training Loss=> 0.0000000, Training ACC 1.0000  

Validation Loss=> 0.00009682, Validation ACC 0.9960  

***->>-----<<<-***  

Epoch : 96  

    Training Loss=> 0.0000000, Training ACC 1.0000  

Validation Loss=> 0.00009705, Validation ACC 0.9960  

***->>-----<<<-***  

Epoch : 97  

    Training Loss=> 0.0000000, Training ACC 1.0000  

Validation Loss=> 0.00009710, Validation ACC 0.9960  

***->>-----<<<-***  

Epoch : 98  

    Training Loss=> 0.0000000, Training ACC 1.0000  

Validation Loss=> 0.00009731, Validation ACC 0.9960  

***->>-----<<<-***  

Epoch : 99  

    Training Loss=> 0.0000000, Training ACC 1.0000  

Validation Loss=> 0.00009747, Validation ACC 0.9960  

***->>-----<<<-***  

Epoch : 100  

    Training Loss=> 0.0000000, Training ACC 1.0000  

Validation Loss=> 0.00009759, Validation ACC 0.9960  

***->>-----<<<-***
```

شکل 95: نتیجه ۱۵ اجرای آخر بدون Data Augmentation

```

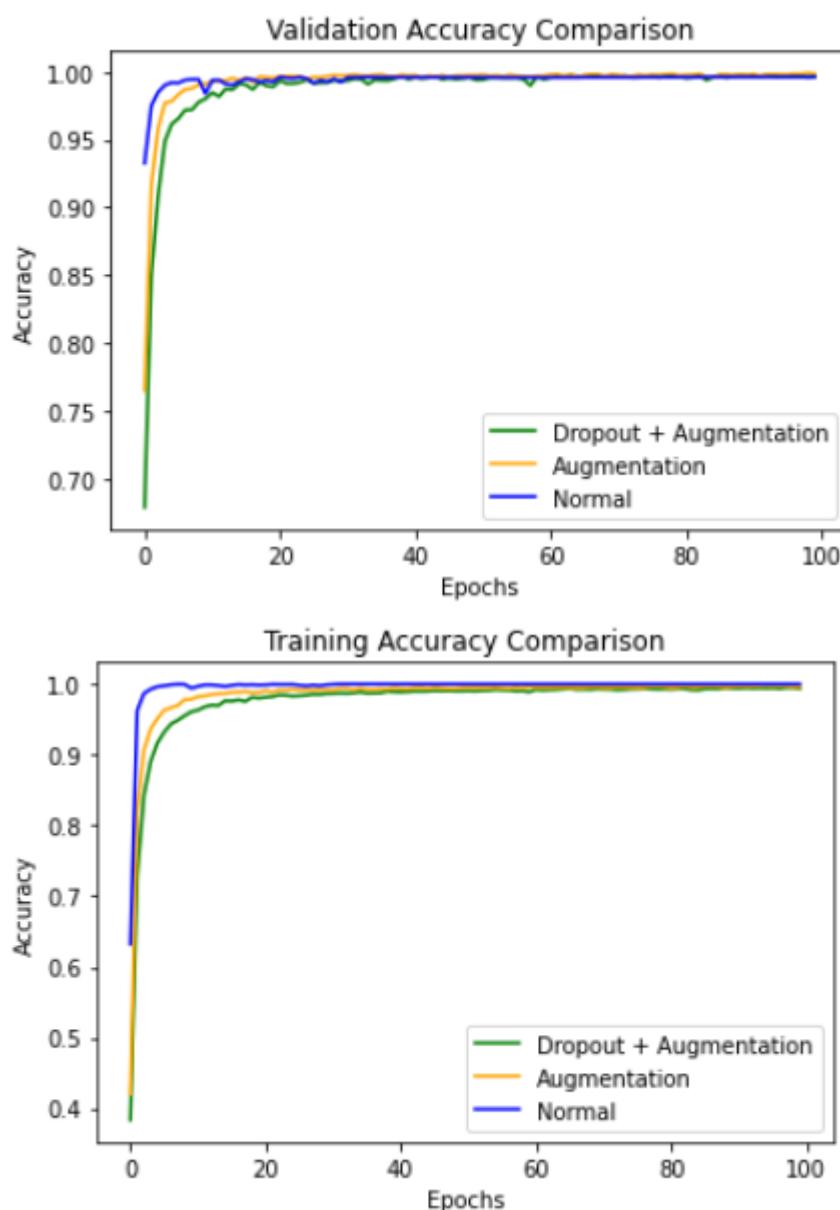
*****  

--->> 3522.8378431797028 seconds <<<---  

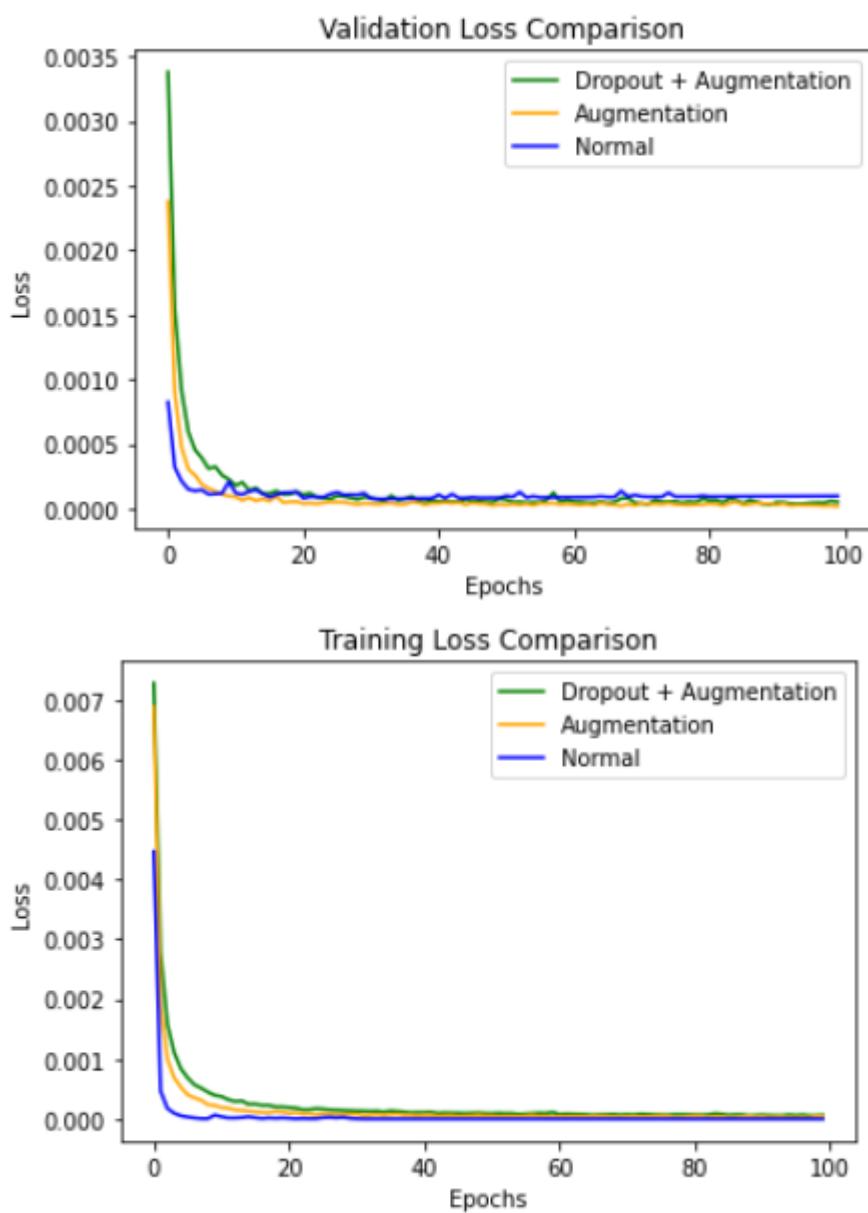
*****
```

شکل 96: زمان اجرای حالت بدون Data Augmentation

سپس در ادامه نمودارهای مربوط به اجرا را قرار می‌دهیم:



شکل ۹۷: نمودار مقایسه Accuracy های ۳ حالت همراه با Data Aug و Dropout و Data Aug و نیز
حالت مدل معمولی ابتدای کار در دو مجموعه داده Train و Test



شکل 98: نمودار مقایسه Loss های 3 حالت همراه با Data Aug و Dropout و Data Aug تنها و نیز حالت مدل معمولی ابتدای کار در دو مجموعه داده Train و Test

➤ نتیجه‌گیری:

- ✓ با مشاهده نمودارهای بالا به این نتیجه می‌رسیم که در هر دو حالت آموزش و ارزیابی نمودار حالت‌های با Data Augmentation به صورت Smooth تری عمل می‌کند و دیرتر به حداقل دقت یا حداقل Loss می‌رسد و علت اصلی آن هم این است که در این دو مدل مجموعه داده‌های ما بسیار بزرگ‌تر است و علاوه بر دیتاست اصلی شامل داده‌های Augment شده نیز

می‌باشند بنابراین فرآیند آموزش کمی کندتر است و زمان بیشتری طول می‌کشد تا شبکه بتواند تمامی این داده‌ها را یاد بگیرد.

✓ در نمودار Loss نیز ما مشاهده می‌کنیم که در حالت‌هایی که ما از Data Augmentation استفاده می‌کنیم، مقدار Loss ما کمتر از حالت عادی نیز می‌شود و بنابراین در زمینه کاهش Loss عملکرد بسیار خوبی از خود نشان می‌دهد.

✓ علت این که ما تاثیر بسیار زیادی از اضافه کردن این موارد به شبکه مشاهده نمی‌کنیم اولاً این است که نیاز به ایپاک‌های بیشتری برای آموزش هست و ثانیاً نیز بهترین عملکرد خود را Data Augmentation و همچنین Dropout در جاهایی به ما نشان می‌دهند که مدل اصلی ما overfit شود که در اینجا پیش نیامده پس نمی‌توانیم انتظار یک بهبود انقلابی‌ای را داشته باشیم.

✓ اما موردی که مشخص است استفاده از Data Augmentation باعث می‌شود در نهایت که شبکه ما داده‌های بیشتری برای آموزش داشته باشد و به این ترتیب بتواند خود را برای داده‌های تست سخت‌تری آماده کند. اما خب همان‌طور که از زمان‌های اجرایی که قرار داده‌ام مشخص است استفاده از این روش کمی زمان آموزش شبکه را نیز به دلیل بیشتر شدن داده‌ها زیاد می‌کند.

• **توجه: فایل مربوط به این سوال در پوشه آپلود شده در فایلی به نام Practical_Part_G_Final.ipynb قرار دارد.**

:H قسمت ❖

سوال : در این قسمت از سوال از ما خواسته شده است که تاثیر اضافه کردن لایه Batch Normalization را در شبکه مورد ارزیابی قرار دهیم.

پاسخ:

من برای پاسخگویی به این سوال یک قسمت دیگر را نیز اضافه کردم و آن هم لایه Drop Out به همراه Normalization.

در مورد مفاهیم و نحوه کارکرد لایه Batch Normalization پیشتر در قسمت سوالات تشریحی پرداختم و به همین دلیل به این موضوعات در اینجا دیگر اشاره‌ای نمی‌کنم.

ابتدا کدهای مربوط به اضافه کردن لایه Batch Normalization را در ادامه قرار می‌دهم:

```
75 class CnnModelNorm(nn.Module):
76     def __init__(self):
77         super(CnnModelNorm, self).__init__()
78         self.layer1 = nn.Sequential(
79             nn.Conv2d(in_channels=3, out_channels=73, kernel_size=5, stride=1, padding=0),
80             nn.BatchNorm2d(73),
81             nn.ReLU(),
82         )
83         self.layer2 = nn.Sequential(
84             nn.Conv2d(in_channels=73, out_channels=153, kernel_size=3, stride=1, padding=0),
85             nn.BatchNorm2d(153),
86             nn.ReLU(),
87             nn.MaxPool2d(kernel_size=2, stride=2),
88         )
89         self.layer3 = nn.Sequential(
90             nn.Conv2d(in_channels=153, out_channels=273, kernel_size=3, stride=1, padding=1),
91             nn.BatchNorm2d(273),
92             nn.ReLU(),
93             nn.MaxPool2d(kernel_size=2, stride=2),
94         )
95         self.fc1 = nn.Linear(6 * 6 * 273, 500) # 1: Stride
96         self.relu = nn.ReLU()
97         self.fc2 = nn.Linear(500, 43) # 1: Stride
98         self.softmax = nn.LogSoftmax(dim=1)
99
100    def forward(self, x):
101        out = self.layer1(x)
102        out = self.layer2(out)
103        out = self.layer3(out)
104
105        out = torch.flatten(out, 1)
106        out = self.fc1(out)
107        out = self.relu(out)
108        out = self.fc2(out)
109        # out = self.softmax(out)
110        return out
```

شکل ۹۹. اضافه کردن لایه‌های Batch Normalization به مدل طراحی شده

```

35 class CnnModelDropNorm(nn.Module):
36     def __init__(self):
37         super(CnnModelDropNorm, self).__init__()
38         self.layer1 = nn.Sequential(
39             nn.Conv2d(in_channels=3, out_channels=73, kernel_size=5, stride=1, padding=0),
40             nn.BatchNorm2d(73),
41             nn.ReLU(),
42         )
43         self.layer2 = nn.Sequential(
44             nn.Conv2d(in_channels=73, out_channels=153, kernel_size=3, stride=1, padding=0),
45             nn.BatchNorm2d(153),
46             nn.ReLU(),
47             nn.MaxPool2d(kernel_size=2, stride=2),
48         )
49         self.layer3 = nn.Sequential(
50             nn.Conv2d(in_channels=153, out_channels=273, kernel_size=3, stride=1, padding=1),
51             nn.BatchNorm2d(273),
52             nn.ReLU(),
53             nn.MaxPool2d(kernel_size=2, stride=2),
54         )
55         self.fc1 = nn.Linear(6 * 6 * 273, 500) # 1: Stride
56         self.relu = nn.ReLU()
57         self.drop1 = nn.Dropout(p=0.5)
58         self.fc2 = nn.Linear(500, 43) # 1: Stride
59         self.softmax = nn.LogSoftmax(dim=1)
60
61     def forward(self, x):
62         out = self.layer1(x)
63         out = self.layer2(out)
64         out = self.layer3(out)
65
66         out = torch.flatten(out, 1)
67         out = self.fc1(out)
68         out = self.relu(out)
69         out = self.drop1(out)
70         out = self.fc2(out)
71         # out = self.softmax(out)
72         return out

```

شکل 100: کد مربوط به اضافه کردن Batch Normalization

سپس این مدل‌های طراحی شده را به همراه مدل اولیه خود به صورت جداگانه مطابق کدهای زیر آموزش می‌دهیم با استفاده از تابعی که پیشتر قرار دادم.

ابتدا من این اعمال را بدون استفاده از Data Augmentation و سپس با استفاده از انجام می‌دهم تا اثر استفاده از این روش را نیز بتوانیم بهتر متوجه شویم.

```

1. from torch.optim.lr_scheduler import MultiStepLR
2.
3.
4. learning_rate = 0.00073
5. epochs = 100
6.
7. criterion = nn.CrossEntropyLoss()
8.
9. # scheduler_relu = MultiStepLR(optimizer_relu, milestones=[30, 70], gamma=0.1)
10. #####
11. #####
12. #####
13. model_drop_norm = CnnModelDropNorm().to(CUDA)

```

```

14. optimizer_drop_norm = torch.optim.Adam(model_drop_norm.parameters(), lr=learning_rate)
15. print("### *** Dropout 1 + Batch Normalization : ===>> *** ###")
16. train_loss_drop_norm, val_loss_drop_norm, train_acc_drop_norm, val_acc_drop_norm = model_training(model=model_drop_norm, epochs=epochs, optimizer=optimizer_drop_norm, scheduler=None, criterion=criterion, train_loader=train_loader)
17. #####
18. model_norm = CnnModelNormal().to(CUDA)
19. optimizer_norm = torch.optim.Adam(model_norm.parameters(), lr=learning_rate)
20. print("### *** Normal + Batch Normalization : ===>> *** ###")
21. train_loss_norm, val_loss_norm, train_acc_norm, val_acc_norm = model_training(model=model_norm, epochs=epochs, optimizer=optimizer_norm, scheduler=None, criterion=criterion, train_loader=train_loader)
22. #####
23. model_normal = CnnModelNormal().to(CUDA)
24. optimizer_normal = torch.optim.Adam(model_normal.parameters(), lr=learning_rate)

25. print("### *** Normal : ===>> *** ###")
26. train_loss_normal, val_loss_normal, train_acc_normal, val_acc_normal = model_training(model=model_normal, epochs=epochs, optimizer=optimizer_normal, scheduler=None, criterion=criterion, train_loader=train_loader)
27. #####

```

پس از اجرای کد فوق، نتایج به صورت زیر به دست می‌آید:

```

### *** Dropout 1 + Batch Normalization : ===>> *** ###
Epoch : 1
    Training Loss:=> 0.00701372,      Training ACC 0.4199
    Validation Loss:=> 0.00254001,   Validation ACC 0.7660
***->>>-----<<<-***

Epoch : 2
    Training Loss:=> 0.00119031,      Training ACC 0.8942
    Validation Loss:=> 0.00055841,   Validation ACC 0.9554
***->>>-----<<<-***

Epoch : 3
    Training Loss:=> 0.00041200,      Training ACC 0.9672
    Validation Loss:=> 0.00042859,   Validation ACC 0.9640
***->>>-----<<<-***

Epoch : 4
    Training Loss:=> 0.00023210,      Training ACC 0.9819
    Validation Loss:=> 0.00023349,   Validation ACC 0.9807
***->>>-----<<<-***

Epoch : 5
    Training Loss:=> 0.00016044,      Training ACC 0.9873
    Validation Loss:=> 0.00020336,   Validation ACC 0.9827
***->>>-----<<<-***
```

شکل 101: نتیجه 5 اجرای اول حالت Data Augmentation بدون Dropout + Batch Normalization

```

***->>-----<<<-***  

Epoch : 95  

    Training Loss=> 0.00001094,      Training ACC 0.9990  

Validation Loss=> 0.00013723,  Validation ACC 0.9968  

***->>-----<<<-***  

Epoch : 96  

    Training Loss=> 0.00000485,      Training ACC 0.9996  

Validation Loss=> 0.00005676,  Validation ACC 0.9976  

***->>-----<<<-***  

Epoch : 97  

    Training Loss=> 0.00000513,      Training ACC 0.9996  

Validation Loss=> 0.00004180,  Validation ACC 0.9976  

***->>-----<<<-***  

Epoch : 98  

    Training Loss=> 0.00000551,      Training ACC 0.9997  

Validation Loss=> 0.00004909,  Validation ACC 0.9974  

***->>-----<<<-***  

Epoch : 99  

    Training Loss=> 0.00000402,      Training ACC 0.9997  

Validation Loss=> 0.00004732,  Validation ACC 0.9971  

***->>-----<<<-***  

Epoch : 100  

    Training Loss=> 0.00000483,      Training ACC 0.9996  

Validation Loss=> 0.00007909,  Validation ACC 0.9969  

***->>-----<<<-***
```

شکل 102: نتیجه 5 اجرای آخر حالت Data Augmentation بدون Batch Normalization + Dropout

```

*****  

--->> 2153.9908254146576 seconds <<<---  

*****
```

شکل 103: زمان اجرای حالت Data Augmentation بدون Dropout + Batch Normalization

```

### *** Dropout 1 + Batch Normalization : ===>> *** ####
Epoch : 1
    Training Loss=> 0.00888609,    Training ACC 0.2682
Validation Loss=> 0.00531119, Validation ACC 0.5099
***->>-----<<<-***

Epoch : 2
    Training Loss=> 0.00371866,    Training ACC 0.6354
Validation Loss=> 0.00216524, Validation ACC 0.7923
***->>-----<<<-***

Epoch : 3
    Training Loss=> 0.00199740,    Training ACC 0.8047
Validation Loss=> 0.00121801, Validation ACC 0.8865
***->>-----<<<-***

Epoch : 4
    Training Loss=> 0.00139236,    Training ACC 0.8656
Validation Loss=> 0.00086601, Validation ACC 0.9188
***->>-----<<<-***

Epoch : 5
    Training Loss=> 0.00111876,    Training ACC 0.8929
Validation Loss=> 0.00073014, Validation ACC 0.9306
***->>-----<<<-***

```

شکل 104: نتیجه 5 اجرای اول حالت **Dropout + Batch Normalization** با استفاده از **Augmentation**

```

***->>-----<<<-***
Epoch : 95
    Training Loss=> 0.00009391,    Training ACC 0.9909
Validation Loss=> 0.00006783, Validation ACC 0.9952
***->>-----<<<-***

Epoch : 96
    Training Loss=> 0.00008789,    Training ACC 0.9916
Validation Loss=> 0.00008057, Validation ACC 0.9941
***->>-----<<<-***

Epoch : 97
    Training Loss=> 0.00008066,    Training ACC 0.9919
Validation Loss=> 0.00007449, Validation ACC 0.9952
***->>-----<<<-***

Epoch : 98
    Training Loss=> 0.00009301,    Training ACC 0.9917
Validation Loss=> 0.00006287, Validation ACC 0.9946
***->>-----<<<-***

Epoch : 99
    Training Loss=> 0.00008730,    Training ACC 0.9915
Validation Loss=> 0.00005357, Validation ACC 0.9955
***->>-----<<<-***

Epoch : 100
    Training Loss=> 0.00008904,    Training ACC 0.9916
Validation Loss=> 0.00010693, Validation ACC 0.9940
***->>-----<<<-***

```

شکل 105: نتیجه 5 اجرای آخر حالت **Batch Normalization + Dropout** با استفاده از **Augmentation**

شکل 106: زمان اجرای حالت Data Augmentation به همراه Dropout + Batch Normalization در 100 ایپاک

```
*****
--->>> 4871.303320407867 seconds <<<---
*****  
....  
### *** Normal + Batch Normalization : ===>>> *** ###  
Epoch : 1  
    Training Loss=> 0.00653750,    Training ACC 0.4429  
Validation Loss=> 0.00226211, Validation ACC 0.7930  
***->>>-----<<<-***  
Epoch : 2  
    Training Loss=> 0.00180446,    Training ACC 0.8289  
Validation Loss=> 0.00083123, Validation ACC 0.9288  
***->>>-----<<<-***  
Epoch : 3  
    Training Loss=> 0.00096042,    Training ACC 0.9112  
Validation Loss=> 0.00050689, Validation ACC 0.9591  
***->>>-----<<<-***  
Epoch : 4  
    Training Loss=> 0.00066434,    Training ACC 0.9371  
Validation Loss=> 0.00031729, Validation ACC 0.9735  
***->>>-----<<<-***  
Epoch : 5  
    Training Loss=> 0.00049196,    Training ACC 0.9544  
Validation Loss=> 0.00027874, Validation ACC 0.9762  
***->>>-----<<<-***
```

شکل 107: نتیجه 5 اجرای اول حالت Batch Normalization تنها با استفاده از Data Augmentation

```

***->>-----<<<-***  

Epoch : 95  

    Training Loss:=> 0.00004326,      Training ACC 0.9961  

    Validation Loss:=> 0.00004842,   Validation ACC 0.9973  

***->>-----<<<-***  

Epoch : 96  

    Training Loss:=> 0.00002745,      Training ACC 0.9973  

    Validation Loss:=> 0.00004643,   Validation ACC 0.9974  

***->>-----<<<-***  

Epoch : 97  

    Training Loss:=> 0.00003350,      Training ACC 0.9970  

    Validation Loss:=> 0.00004025,   Validation ACC 0.9982  

***->>-----<<<-***  

Epoch : 98  

    Training Loss:=> 0.00003359,      Training ACC 0.9970  

    Validation Loss:=> 0.00003542,   Validation ACC 0.9978  

***->>-----<<<-***  

Epoch : 99  

    Training Loss:=> 0.00002514,      Training ACC 0.9978  

    Validation Loss:=> 0.00002975,   Validation ACC 0.9990  

***->>-----<<<-***  

Epoch : 100  

    Training Loss:=> 0.00003765,      Training ACC 0.9966  

    Validation Loss:=> 0.00008975,   Validation ACC 0.9964  

***->>-----<<<-***
```

شکل 108: نتیجه 5 اجرای آخر حالت Data Augmentation بعدها Batch Normalization

```

### *** Normal + Batch Normalization : ==>> *** ###  

Epoch : 1  

    Training Loss:=> 0.00491064,      Training ACC 0.5931  

    Validation Loss:=> 0.00090262,   Validation ACC 0.9286  

***->>-----<<<-***  

Epoch : 2  

    Training Loss:=> 0.00044839,      Training ACC 0.9640  

    Validation Loss:=> 0.00037675,   Validation ACC 0.9726  

***->>-----<<<-***  

Epoch : 3  

    Training Loss:=> 0.00017250,      Training ACC 0.9868  

    Validation Loss:=> 0.00018254,   Validation ACC 0.9862  

***->>-----<<<-***  

Epoch : 4  

    Training Loss:=> 0.00007994,      Training ACC 0.9943  

    Validation Loss:=> 0.00017404,   Validation ACC 0.9867  

***->>-----<<<-***  

Epoch : 5  

    Training Loss:=> 0.00005237,      Training ACC 0.9963  

    Validation Loss:=> 0.00012255,   Validation ACC 0.9918  

***->>-----<<<-***
```

شکل 109: نتیجه 5 اجرای اول حالت Data Augmentation بدون Batch Normalization

```

***->>-----<<<-***  

Epoch : 95  

    Training Loss=> 0.00000000,     Training ACC 1.0000  

Validation Loss=> 0.00010136, Validation ACC 0.9968  

***->>-----<<<-***  

Epoch : 96  

    Training Loss=> 0.00000000,     Training ACC 1.0000  

Validation Loss=> 0.00010150, Validation ACC 0.9968  

***->>-----<<<-***  

Epoch : 97  

    Training Loss=> 0.00000000,     Training ACC 1.0000  

Validation Loss=> 0.00012015, Validation ACC 0.9968  

***->>-----<<<-***  

Epoch : 98  

    Training Loss=> 0.00000000,     Training ACC 1.0000  

Validation Loss=> 0.00010180, Validation ACC 0.9968  

***->>-----<<<-***  

Epoch : 99  

    Training Loss=> 0.00000000,     Training ACC 1.0000  

Validation Loss=> 0.00010184, Validation ACC 0.9968  

***->>-----<<<-***  

Epoch : 100  

    Training Loss=> 0.00000000,     Training ACC 1.0000  

Validation Loss=> 0.00010737, Validation ACC 0.9968  

***->>-----<<<-***
```

شکل 110: نتیجه 5 اجرای آخر حالت تنها بدون Batch Normalization

```

*****  

--->>> 2002.0365991592407 seconds <<<----  

*****
```

شکل 111: زمان اجرای 100 اپیک حالت تنها بدون Batch Normalization

```

### *** Normal : ===>> *** ####
Epoch : 1
    Training Loss:=> 0.00433399,    Training ACC 0.6417
Validation Loss:=> 0.00096607, Validation ACC 0.9263
***->>>-----<<<-***

Epoch : 2
    Training Loss:=> 0.00045628,    Training ACC 0.9645
Validation Loss:=> 0.00027750, Validation ACC 0.9815
***->>>-----<<<-***

Epoch : 3
    Training Loss:=> 0.00016958,    Training ACC 0.9877
Validation Loss:=> 0.00016470, Validation ACC 0.9871
***->>>-----<<<-***

Epoch : 4
    Training Loss:=> 0.00007511,    Training ACC 0.9945
Validation Loss:=> 0.00013722, Validation ACC 0.9902
***->>>-----<<<-***

Epoch : 5
    Training Loss:=> 0.00005584,    Training ACC 0.9955
Validation Loss:=> 0.00012904, Validation ACC 0.9916
***->>>-----<<<-***
```

شکل 112: نتیجه ۵ اجرای اول حالت عادی

```

***->>>-----<<<-***
Epoch : 95
    Training Loss:=> 0.00000000,    Training ACC 1.0000
Validation Loss:=> 0.00012799, Validation ACC 0.9953
***->>>-----<<<-***

Epoch : 96
    Training Loss:=> 0.00000000,    Training ACC 1.0000
Validation Loss:=> 0.00012831, Validation ACC 0.9954
***->>>-----<<<-***

Epoch : 97
    Training Loss:=> 0.00000000,    Training ACC 1.0000
Validation Loss:=> 0.00012879, Validation ACC 0.9952
***->>>-----<<<-***

Epoch : 98
    Training Loss:=> 0.00000000,    Training ACC 1.0000
Validation Loss:=> 0.00012874, Validation ACC 0.9953
***->>>-----<<<-***

Epoch : 99
    Training Loss:=> 0.00000000,    Training ACC 1.0000
Validation Loss:=> 0.00012862, Validation ACC 0.9954
***->>>-----<<<-***

Epoch : 100
    Training Loss:=> 0.00000000,    Training ACC 1.0000
Validation Loss:=> 0.00020469, Validation ACC 0.9953
***->>>-----<<<-***
```

شکل 113: نتیجه ۵ اجرای آخر حالت عادی

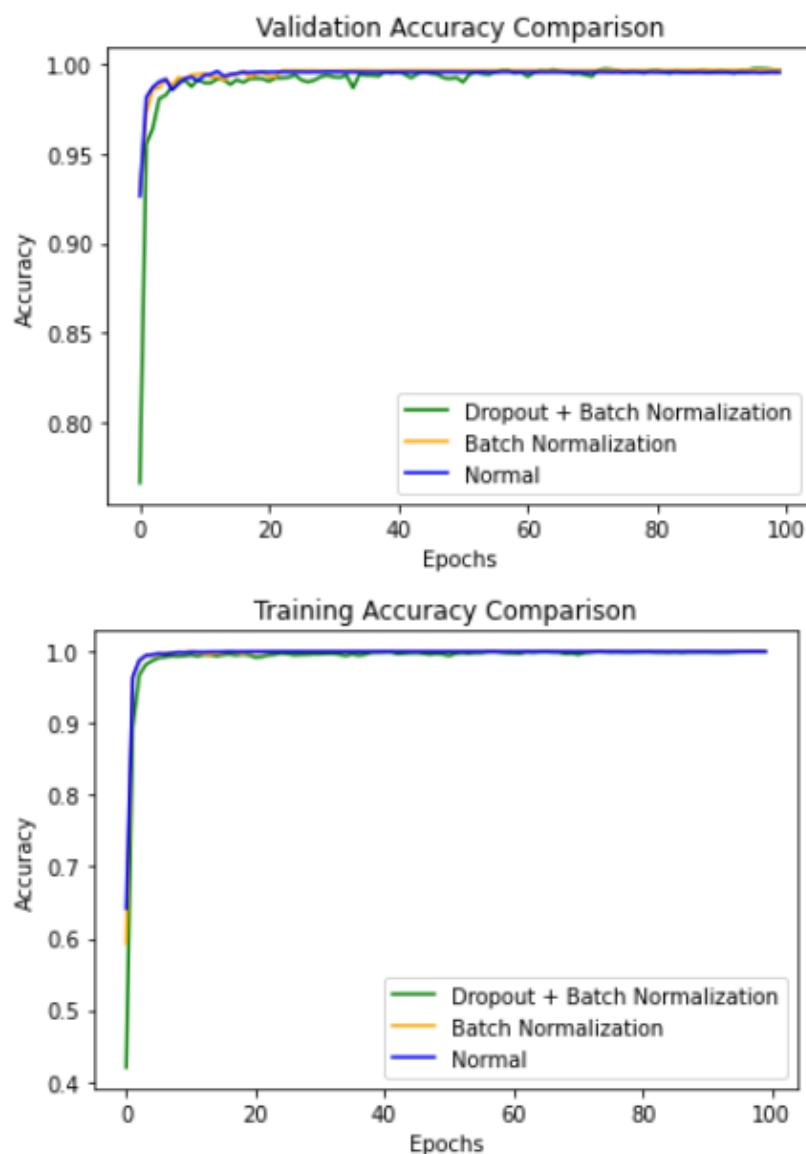
```

*****
-->>> 2010.943753004074 seconds <<<---
*****

```

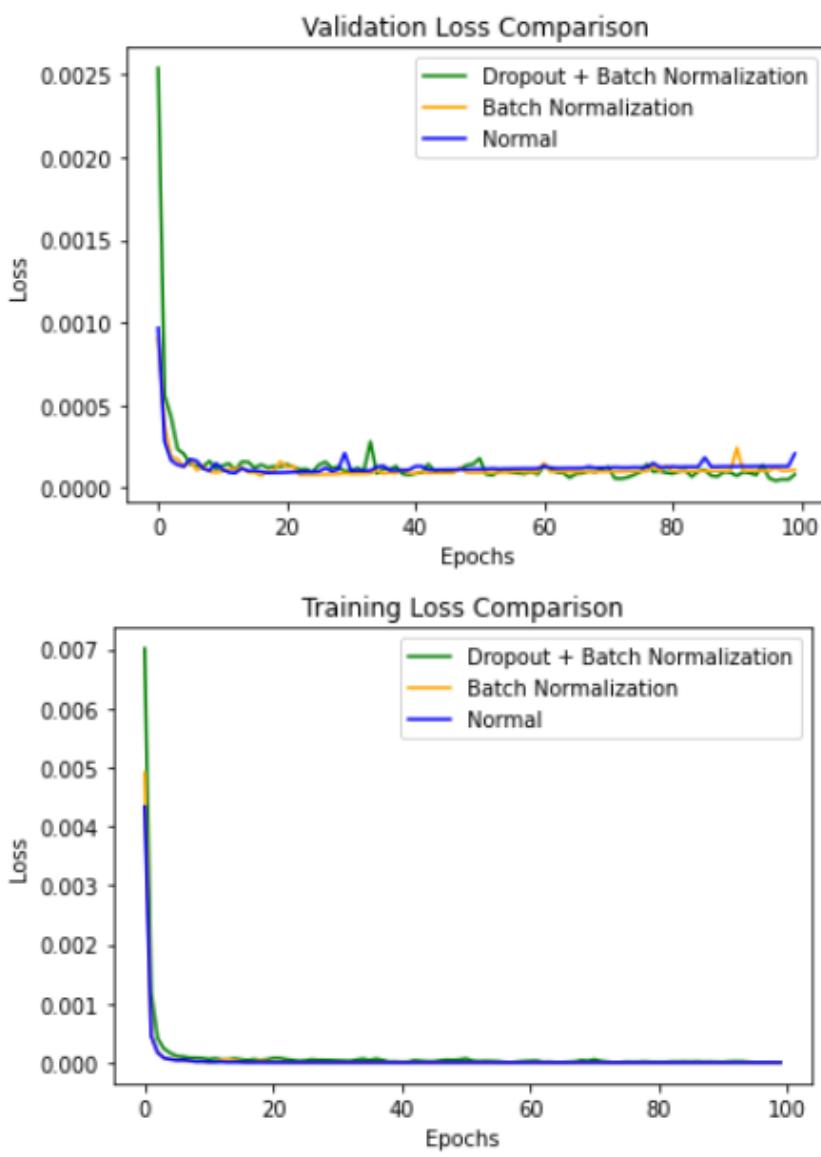
شکل 114: زمان اجرای حالت عادی

سپس نمودارهای اجراها نیز مطابق شکل های زیر می شوند:



شکل 115: نمودار مقایسه داده های Validation و Train Accuracy بین حالت های Batch و Normal

Dropout و Batch Normalization همراه با Normalization



شکل 116: نمودار مقایسه Loss داده‌های Train و Validation بین حالت‌های Normal و Batch Normalization همراه با Dropout

```

#### *** Dropout + Batch Normalization : ===>> *** ####
*****
Test Loss: 0.00058913, Test ACC: 0.9739
*****
#### *** Batch Normalization : ===>> *** ####
*****
Test Loss: 0.00168582, Test ACC: 0.9591
*****
#### *** Normal : ===>> *** ####
*****
Test Loss: 0.00216874, Test ACC: 0.9518
*****

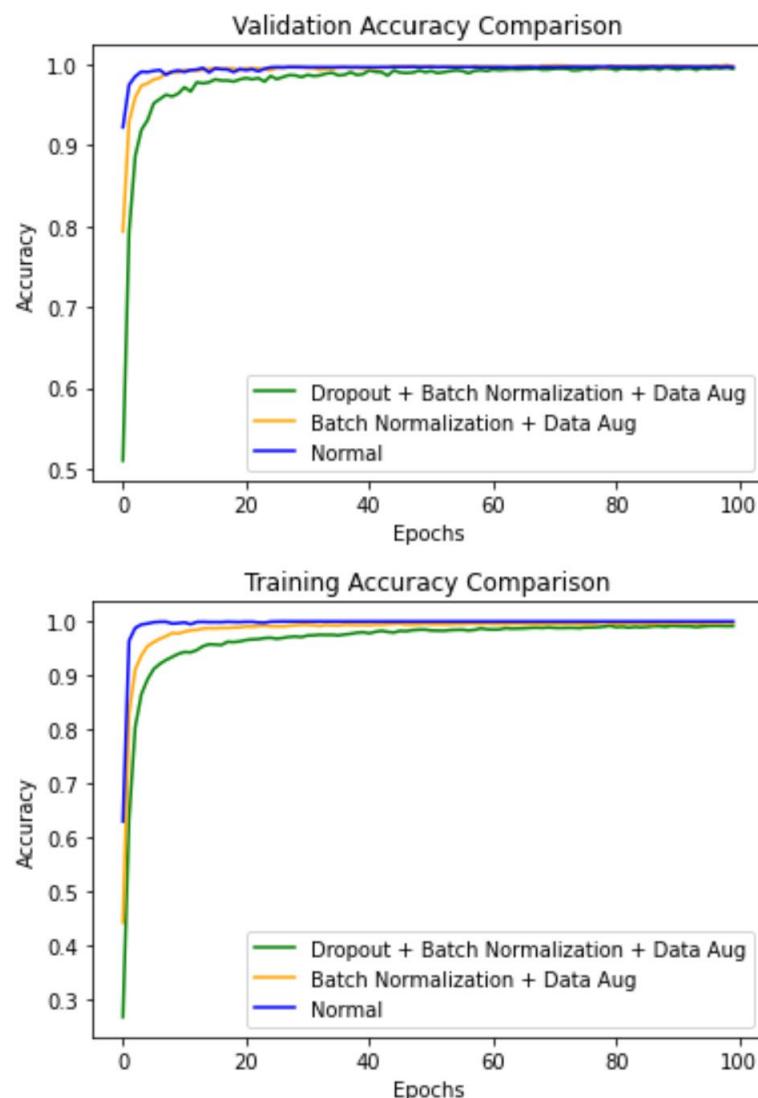
```

شکل 117: دقت‌های به دست آمده در سه حالت بالا بر روی داده‌های Test

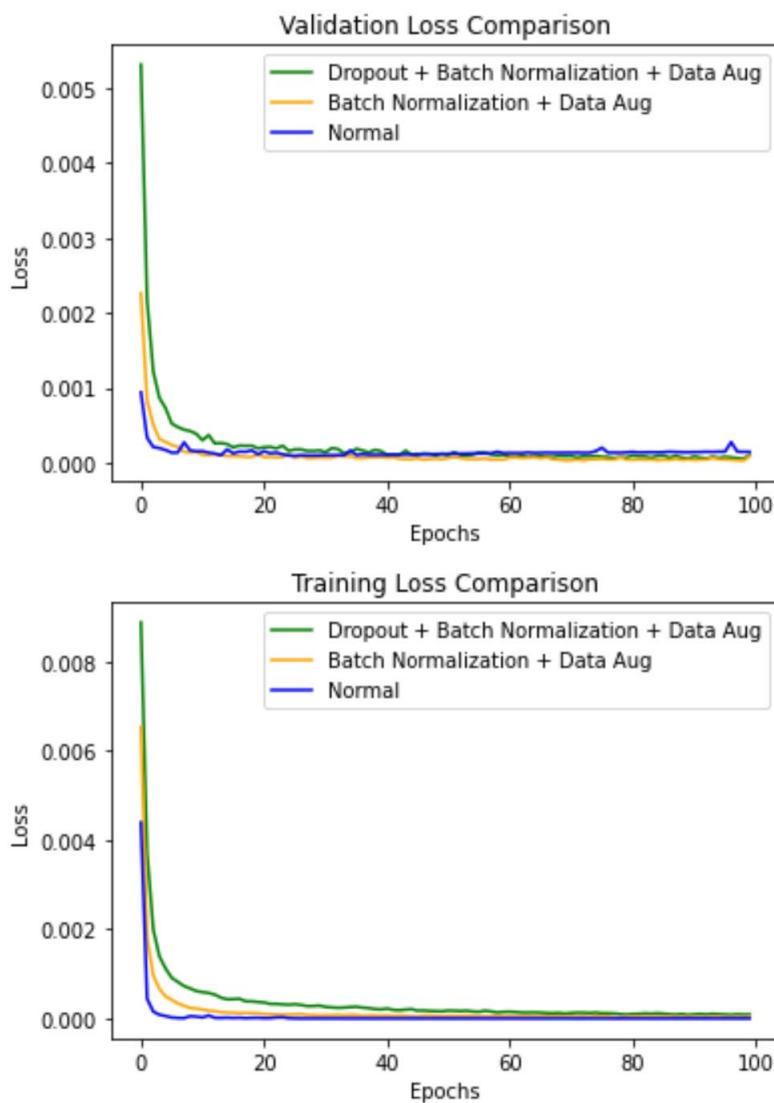
► نتیجه‌گیری:

- ✓ همان‌طور که از دقت‌های به دست آمده بر روی داده‌های تست مشخص است، بهترین عملکرد را ترکیب Dropout و Batch Normalization داشته است. و پس از آن Normalization تنها و سپس هم مدل اولیه‌ای که داشتیم.
- ✓ علت خوب عمل کردن Dropout را که در قسمت مربوطه بررسی کردیم. اما نکته اضافه در اینجا Batch Normalization است. کاری که Batch Normalization انجام می‌دهد این است که می‌آید و در هر لایه با توجه به Batch داده شده به آن و برای این که مقدار نرون‌ها زیاد نشود، اقدام به نرمال‌سازی آن Batch می‌کند و من توضیحات بیشتر و فرمول و نحوه کار آن را در سوال آخر قسمت تشریحی قرار داده‌ام. انجام این کار باعث می‌شود که شبکه به سمت به ویژگی خاص نرود و یک ویژگی خاصی را فقط خوب یاد نگیرد و بین همه ویژگی‌ها تعادل برقرار کند.
- ✓ از بررسی نمودارها مشخص است که در ابتدای کار کمی عملکرد حالت Batch Normalization به دلیل این که در ابتدای خیلی نیازی به نرمال‌سازی نیست و نرمال‌سازی باعث کاهش دقت شده است اما در epoch های بعدی که مقادیر نرون‌ها زیاد می‌شود این نیاز وجود دارد و با انجام این کار باعث بهبود عملکرد شبکه می‌شویم و نمودار ما Smooth تر می‌شود.
- ✓ همچنین از بررسی نمودار Loss به این نتیجه می‌رسیم که مقدار Loss در این نمودارها با شبکه Smooth تری نسبت به حالت عادی پیش رفته و در نهایت مقدار Loss از مقدار در مدل اولیه ما کمتر نیز شده است.

در ادامه نیز نمودارهای مربوط به ترکیب Data Augmentation با خواسته این سوال را قرار می‌دهم
که فراتر از خواسته سوال است اما بررسی آن خالی از لطف نیست:



شکل 118: نمودار مقایسه Accuracy ها در این حالت در دو مجموعه آموزش و ارزیابی



شکل 119: نمودار Loss در این حالت برای دو مجموعه آموزش و ارزیابی

✓ از مشاهده این شکل‌ها می‌توانیم مشاهده کنیم که مانند حالت‌های قبل نمودار در حالت‌های دارای Smooth و Batch Normalization و Dropout و Augmentation کمی تراست و با بررسی که من انجام دادم برای رسیدن به حداقل کارایی نیاز به اجرای این موارد در حدود 300 ایپاک هست که متأسفانه وسط اجرای من به مشکل خورد و نتوانستم تا بهترین نتیجه ممکن را از آن‌ها به دست بیاورم. اما این که این موارد در حال عملکرد بهتری هستند معلوم است و حالت قبل که Data Augmentation هم نداشتیم توانستم با 100 ایپاک دقیقی حدود 97.5 درصد را بگیرم که خیلی خوب هست اما با اضافه شدن Data Augmentation خب حجم داده‌ها و محاسبات زیادتر می‌شود و شبکه نیاز به epohs بیشتری دارد تا به بهترین نتیجه در آموزش خود برسد.

✓ به علاوه همان طور که در حالت‌های قبل هم گفتم ما تنها عملکرد انقلابی و Dropout Data را در جاهایی می‌بینیم که مدل ما overfit شده باشد که در این سوال برای من اصلاً پیش نیامد.

- توجه: فایل‌های این قسمت سوال در پوشش باگذاری شده در قسمت دو فایل در و Practical Part و Practical_Part_H_Data_Augmentation.ipynb قرار دارد. Practical_Part_H_Final.ipynb

• در نهایت امیدوارم که گزارش بنده مورد توجه شما قرار بگیرد و اگر کاستی در آن وجود دارد ببخشید، من نهایت سعی خود را کردم که در وقت موجود کاملترین گزارش را بنویسم و به تمامی قسمت‌ها به صورت کامل پاسخ دهم.

باتشکر از زحمات شما.