به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین ششم

نام و نام خانوادگی	آناهیتا هاشم زاده – پرهام بیچرانلویی
شماره دانشجویی	۸۱۰۱۰۰۵۰۲ - ۸۱۰۱۰۰۳۰۳
تاریخ ارسال گزارش	۱۴۰۱.۱۱.۰۷

فهرست

شکل‌ها

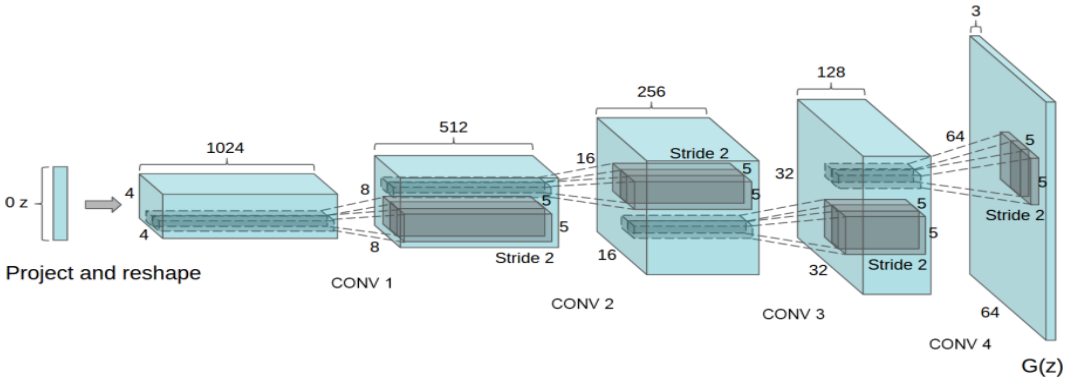
جدول‌ها

پاسخ ۱. شبکه‌های مولد تخاصمی کانولوشنال عمیق

۱-۱. پیاده سازی مولد تصویر با استفاده از شبکه‌های مولد تخاصمی کانولوشنال عمیق

- توضیح مختصر درباره GAN: یک فریمورک برای یادگیری عمیق است. که با بدست آوردن توزیع داده ورودی بتواند داده جدید با توزیع یکسان تولید کند. برای این کار از دو مدل مجزا به نام‌های generator و discriminator استفاده می‌کند. وظیفه generator تولید عکس(داده) فیک مشابه تصاویر واقعی است. وظیفه discriminator هم تشخیص دادن عکس واقعی و عکس فیک است. در زمان train، بخش generator تلاش می‌کند که با تولید عکس‌های مشابه واقعی بخش discriminator رو فریب دهد. در حالیکه بخش discriminator هم با آموزش دیدن تلاش می‌کند که داده‌های واقعی و فیک رو درست تمییز دهد.
- توضیح مختصر درباره DGAN: این شبکه هم یک نسخه از GAN است که معماری generator و discriminator آن متشکل از لایه های convolutional و convolutional-transpose هستند.
که ورودی discriminator یک تصویر RGB 3*64*64 هست و خروجی آن یک عدد است که احتمال اینکه ورودی عکس واقعی باشد را نشان می‌دهد.
ورودی generator هم یک بردار پنهان است که از توزیع نرمال تبعیت می‌کند و خروجی آن یک عکس RGB 3*64*64 هست.
- ملاحظات پیاده سازی:
    - محیط اجرا: Google Colab
    - پردازنده: GPU-T4
    - کتابخونه‌های اصلی: torch- torchvision – matplotlib – random – numpy
    - نام کد: Q2_DCGAN.ipynb
- دیتاست: داده را با ماژول ImageFolder از کتابخونه torchvision می‌خوانیم. تصاویر ما 32*32 هستند. اما معماری ما همانطور که گفته شد بر اساس ورودی 64*64 کار می‌کند برای همین هنگام خواندن دیتاست از یک transform.Resize استفاده میکنیم تا تصویر را 64*64 کند. همچنین برای نتیجه گرفتن بهتر عکس‌ها را نرمالیاز هم می‌کنیم. سپس برای آن یک dataloader می‌سازیم. که سایز batch آن 16 است و همچنین آن را shuffle می‌کنیم. تا اینجا دیتا آماده دادن به مدل شده است.

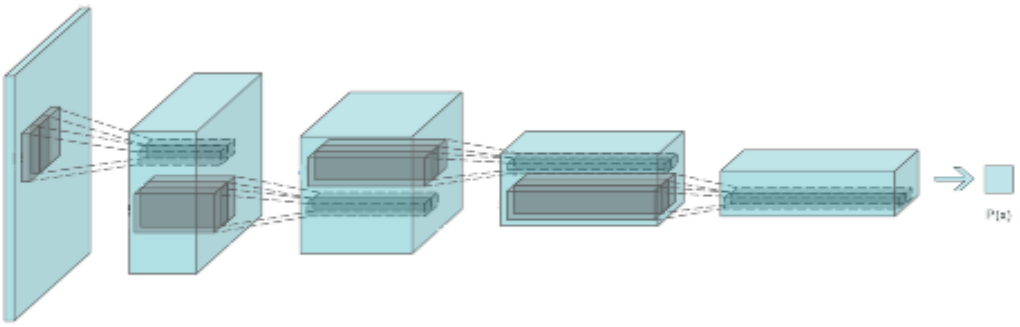
- در مقاله گفته شده است وزن‌های اولیه باید تصادفی باشد و از یک توزیع نرمال با میانگین 0 و انحراف معیار 0.2 باشد. این کار را در تابع weights_init(m) انجام دادم.
- پیاده سازی generator:



شکل ۱. معماری **generator**

این معماری ۵ بلوک متوالی دارد. که هر بلوک شامل یک لایه convolutional transpose ، یک لایه Batch norm و تابع فعال ساز Relu استفاده می‌کند. البته بلوک آخر بعد از conv transpose از تابع فعال ساز tanh استفاده می‌کند. این کار ها را در کلاس generator انجام دادیم. سپس با این کلاس به راحتی یک generator می‌سازیم. و وزن‌های تصادفی اولیه را به آن اعمال می‌کنیم. تا الان مدل generator آماده آموزش است.

- پیاده سازی discriminator:



شکل ۲ . معماری **discriminator**

این معماری هم ۵ بلوک متوالی دارد. که به آرومی اندازه تصویر رو کوچک‌تر می‌کند. به جاش تعداد کانال‌ها رو افزایش میده. هر بلوک شامل یک لایه convolutional و یک لایه batch norm است و در ادامه آن یک تابع فعال ساز LeakyRelu استفاده می‌شود. برای لایه آخر بعد از لایه کانولوشنال بدون نرمالایز کردن تابع sigmoid برای فعال ساز استفاده می‌شود. این کلاس را

فراخوانی می‌کنیم تا یک discriminator بسازیم. و بعد از آن وزن‌های تصادفی اولیه را روی آن اعمال می‌کنیم تا این مدل هم آماده آموزش شود.

- تابع loss: از مفهوم تابع معروف Binary Cross Entropy استفاده می‌کنیم که به صورت زیر تعریف می‌شود:

$$\min_G \max_D V(D,G) = E_x[\log D(x)] + E_Z[\log\left(1 - D\left(G(z)\right)\right)]$$

که log D(x) به احتمال اینکه discriminator عکس واقعی را درست طبقه بندی کند اشاره می‌کند و log (1-D(G(z)) به طبقه بندی درست عکس‌های فیک توسط آن اشاره دارد. برای همین می‌خواهد جمع این مقادیر را زیاد کند.

از آن طرف generator از قسمت دوم فقط استفاده می‌کند و دوست دارد این مقدار کمینه شود یعنی discriminator نتواند عکس‌های تولید شده توسط آن را تشخیص دهد.

- بهینیه ساز: از بهنیه ساز Adam استفاده برای هر دو بخش استفاده می‌کنیم. نرخ یادگیری را 0.00025 در نظر می‌گیریمو بتا را بین رنج 0.5 تا 0.999 در نظر می‌کنیم.
- آموزش شبکه: از دو قسمت تشکلیل شده است. ابتدا بخش discriminator را آموزش می‌دهیم. سپس بخش generator را آموزش میدیم.
  - آموزش discriminator: هدف این است که دقت تشخیض درست عکس واقعی و فیک را افزایش بدهیم. از بهینه ساز adam هم برای آپدیت کردن وزن‌ها استفاده می‌کنیم. داده‌ها را batch به batch آموزش می‌دهیم. به این صورت که ابتدا داده‌های واقعی را با لیبل 1 به discriminator می‌دهیم و loss را حساب کرده و سپس عملیات backward را برای پارامترهای آن انجام می‌دهیم.

    سپس از طریق generator فعلی داده فیک از نویز می‌سازیم. و از discriminator آن را عبور می‌دهیم. مقدار loss را حساب می‌کنیم و برای پارامترهای discriminator عملیات backward زده و وزن هایش را آپدیت می‌کنیم. توجه شود که وزن‌های generator را در این مرحله فریز می‌کنیم یعنی در backward دخالت نمی‌دهیم.
  - آموزش generator: هدف این است که تلاش کنیم داده فیک بهتر و نزدیک‌تر به داده واقعی بسازیم. به دلیل مشکل نداشتن گرادیان کافی به خصوص در گام های اول ، به جای کم کردن log(1-D(G(Z))) مقدار D(G(z)) را بیشینه می‌کنیم.

    همانطور که در کد آمده است از discriminator بخش قبل برای طبقه بندی داده تولید شده توسط generator استفاده می‌کنیم. و مقدار loss را برای generator حساب می‌کنیم. و حالا با backward مقدار وزن‌های generator را به روز می‌کنیم.

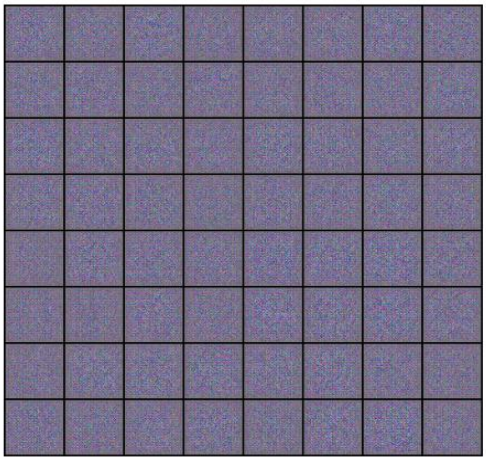
این کار را انقدر ادامه می‌دهیم که داده‌های فیک به اندازه مورد رضایتمان شبیه داده‌های واقعی شوند. در پایان هر ایپاک و iteration هم مقدار loss هر دو را برای گزارش کردن ذخیره می‌کنیم. همچنین بعد هر ایپاک تصاویری را توسط generator تولید کرده تا آن را ارزیابی کنیم.

نکته: لیبل تمام داده‌های فیک را صفر و داده‌های واقعی را یک در نظر می‌گیریم. این کار را با دستور torch.full(batch_size, real_label) انجام می‌دهیم.

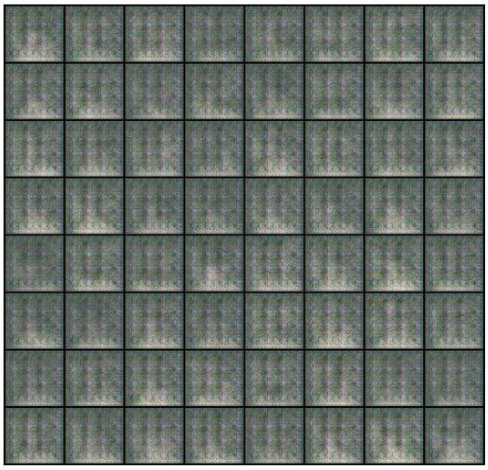
نکته: نویز را از یک توزیع نرمال با دستور torch.randn می‌سازیم.

تصاویر تولید شده توسط generator در یک انیمیشن برای هر ایپاک در کد موجود می‌باشد. اینجا صرفا عکس‌های تولید شده در چند ایپاک ابتدایی و ایپاک آخر را می‌آوریم که در عکس‌های ۳ تا ۶ در ادامه آورده شده است.

ابتدا یک نویز بی معنی را تولید کرده و به مدل generator می‌دهیم که در شکل ۳ آمده است. بعد از آن که یک ایپاک آموزش دید تصاویر محوی دیده میشه. بعد از ۳ ایپاک تصاویر پیشرفت قابل ملاحظه‌ای دارند. و سرانجام بعد از ۵۰ ایپاک این تصاویر قابل رقابت با تصاویر واقعی هستند.

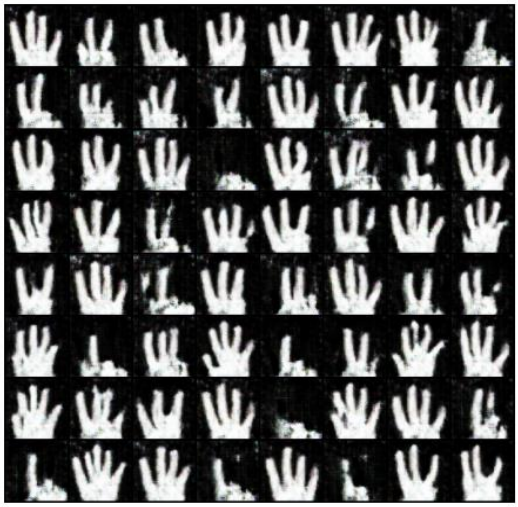


شکل ۳. نویز ورودی



شکل ۴. تصاویر تولید شده بعد ایپاک ۱

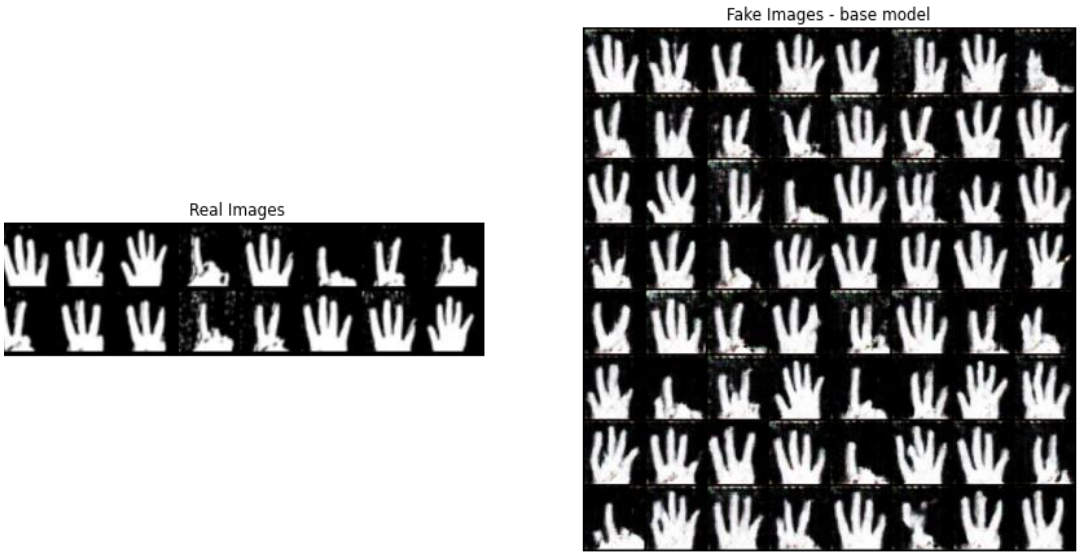
شکل ۵. تصاویر تولید شده بعد ایپاک ۳



شکل ۶. تصاویر تولید شده بعد ایپاک ۵۰

همچنین برای مقایسه بهتر تصویر تولید شده در آخرین ایپاک را کنار تصاویر واقعی می‌گذاریم که در شکل زیر آمده است. که اگر دقت کنیم قابل قبول است اما یکسری ناهنجاری دارند مثل کج بودن، اندازه نامتناسب در برخی عکس‌های تولید شده و ....

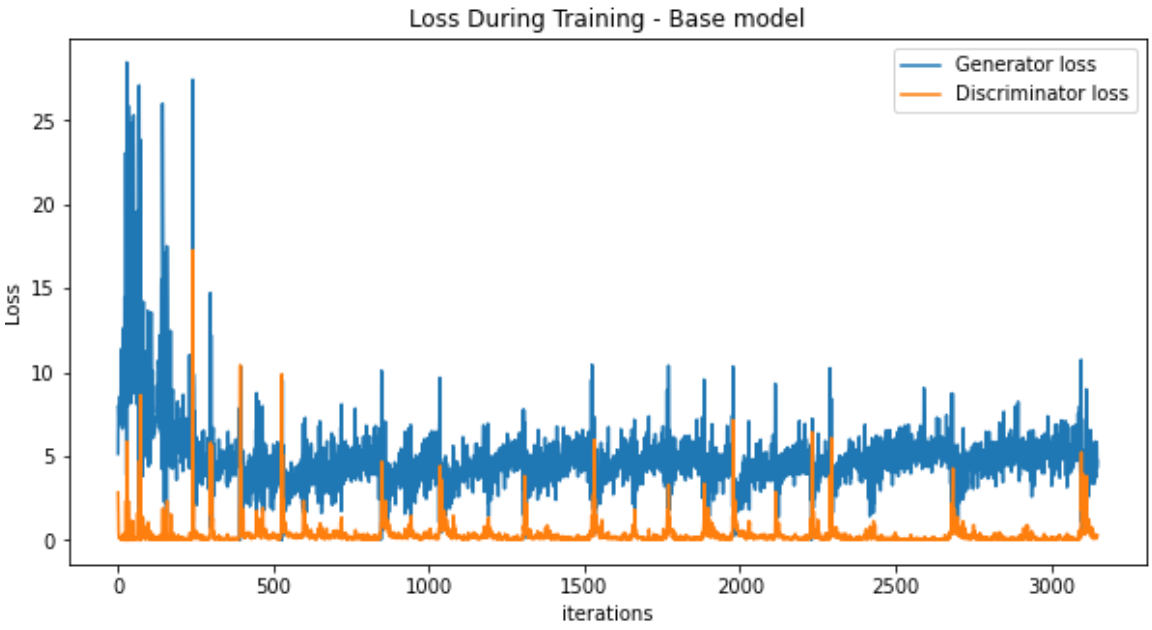
اما کیفیت عکس‌ها نظیر رنگ خیلی خوب است.



شکل ۷. تصاویر فیک تولید شده در مقابل تصاویر واقعی

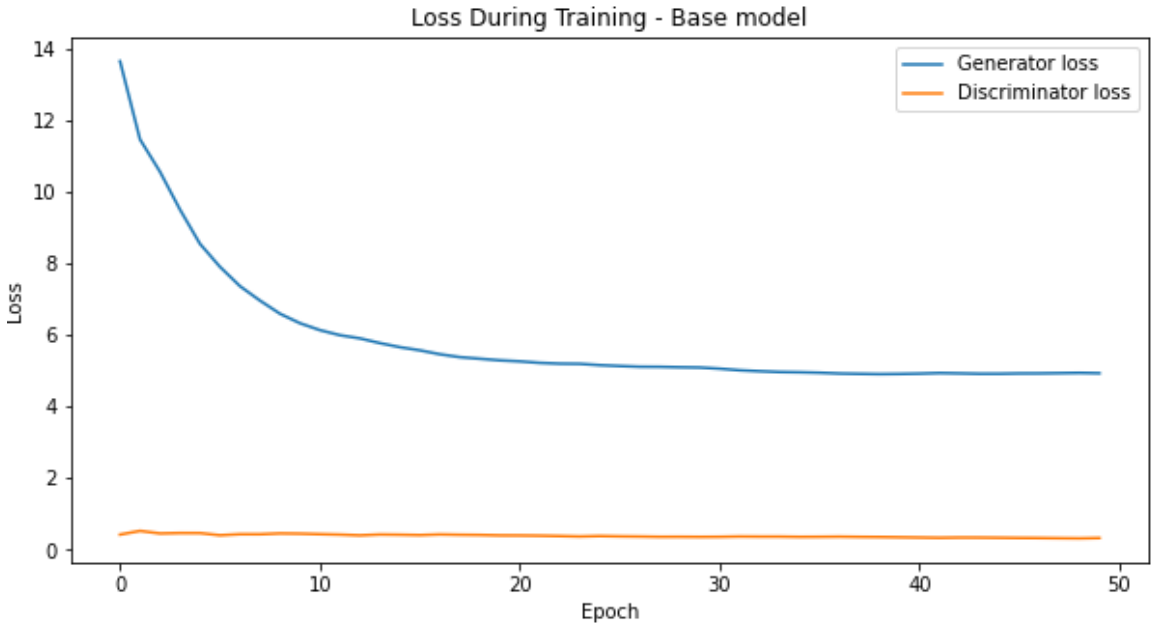
۲-۱. ارزیابی شبکه

نمودار loss برای دو حالت آورده شده است. حالت اول مربوط به مقدار loss محاسبه شده بعد از هر iteration است:



شکل ۸. نمودار **loss** در طول **iterations**

حالت بعدی نمودار loss برای هر ایپاک است که از میانگین loss های iterationهای مربوط به هر ایپاک بدست آورده شده است:



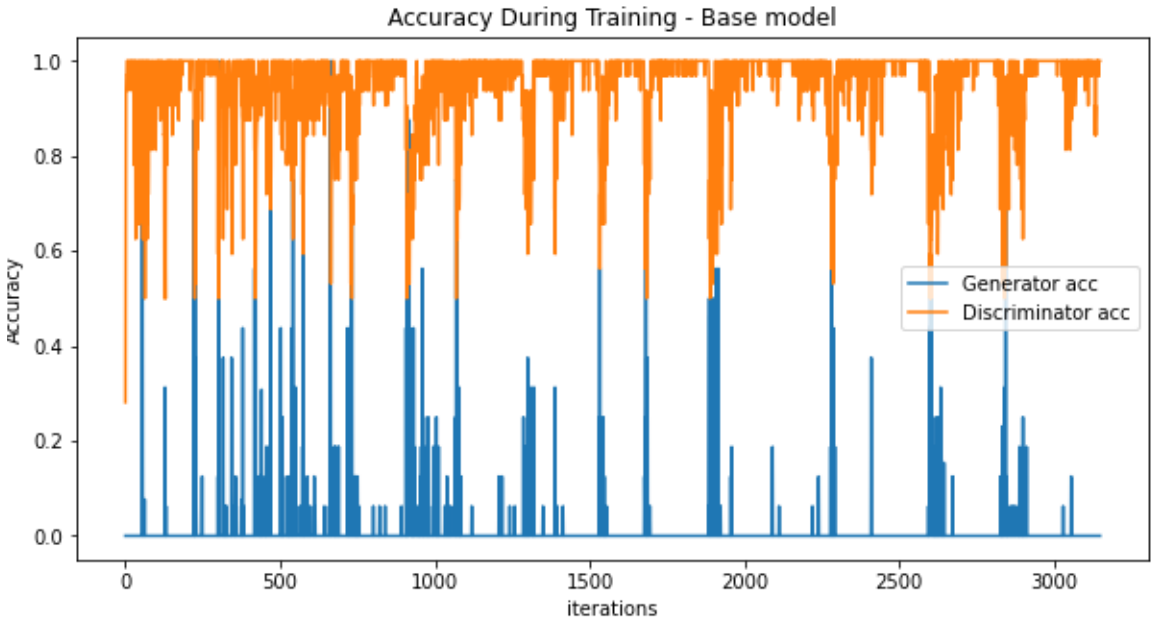
شکل ۹. نمودار **loss** در طول ایپاک‌ها

همانطور که مشاهده می‌کنید مقدار loss برای بخش generator در طول زمان کاهش یافته و به عدد تقریبی ۶ رسیده است. و بعد از ۳۰ ایپاک اشباع شده است. اما نمودار loss برای discriminator تغییر خاصی نداشته است. و این یعنی discriminator ما بهتر عمل می‌کند. تفسیری که از این دو loss می‌توان داشت این است که مدل generator در طول زمان عکس‌های نزدیک‌تر به واقعیت را تولید می‌کند اما از یک حدی نمی‌تواند فرارود. اما چرا این اتفاق می‌افتد؟

زمانی که discriminator خیلی خوب عمل می‌کند، آنوقت generator به دلیل vanishing gradient در آموزش شکست می‌خورد. یک discriminator بهینه اطلاعات کافی را برای generator فراهم نمی‌کند تا بتواند پیشرفت کند. وقتی از backpropagation استفاده می‌کنیم، از قانون ضرب زنجیره‌ای مشتق‌ها استفاده می‌کنیم که اثر چند برابر دارد. یعنی اگر گرادیان کوچک باشد در لایه‌های ابتدایی آنقدر کوچک می‌شود که باعث می‌شود که یادگیری بسیار کند یا متوقف شود.

برای این مشکل چندین راه حل وجود دارد مثل استفاده از wassestein loss یا modified minimax loss. البته استفاده از تابع فعال سازهایی مثل ReLU هم به دلیل مشتق ثابت آن‌ها کمک می‌کند که ما این رو در کدمون پیاده سازی کردیم ولی برای حل مشکل کافی نبوده است.

نمودار دقت را هم در طول iterations در زیر آوردیم:



شکل ۱۰. نمودار دقت در طول **iterations**

در اینجا هم مشخص است که دقت discriminator بهتر است. نوسانی بودن آن می‌تواند به این دلیل باشد که دیتاست ما نسبت به شبکه کوچک است. البته سایز کوچک batch هم بی تاثیر نیست.

۳-۱. پایدارسازی شبکه

تکنیک one-sided label: برای افزایش پایداری شبکه و بهبود عملکرد آن استفاده می‌شود. برای پیاده سازی آن برچسب داده را تغییر می‌دهیم. به جای اینکه به داده واقعی برچسب یک بدهیم به آن برچسب 0.85 می‌دهیم. کمک می‌کند که overconfidence را برای generator در تولید داده‌های واقعی نما کاهش دهیم. این باعث میشه که شبکه پایدارتر شود. در پیاده سازی در کد کافی است به جای اینکه به torch.full لیبل 1بدهیم لیبل 0.85 دهیم. فقط باید توجه شود که وقتی می‌خواهیم دقت را بدست آوریم آستانه را به جای 0.5 باید 0.425 در نظر بگیریم تا دقت بدست آمده معتبر باشد.

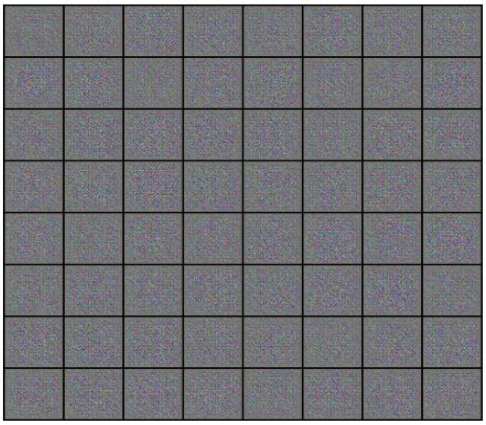
تکنیک add noise: یک نویزی را تولید می‌کنیم و به داده واقعی اضافه می‌کنیم. که این به عنوان یک regularization عمل می‌کند و از overfitting کردن روی داده train جلوگیری می‌کند. و کمک می‌کنه که generator تصاویر گوناگون اما واقعی نما تولید کند.

توضیح بیشتر: در واقع نویز برای شکستن تقارن استفاده میشه. در GAN مدل‌های discriminator و generator معمولا متقارن هستند. یعنی generator می‌تواند عکس‌های فیکی تولید کند که discriminator نتواند از داده‌های واقعی تمیز دهد. با اضافه کردن نویز به عکس‌های واقعی زمان آموزش

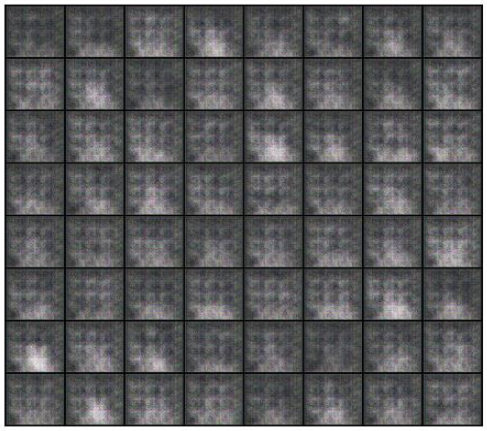
در واقع ما generator را مجبور می‌کنیم که یک representation پیچیده‌تر را از داده یاد بگیرد. به جای اینکه داده‌های train را صرفا حفظ کند. این کمک می‌کند که کیفیت عکس‌های تولید شده بهبود باید و مدل GAN را مقاوم‌تر کند.

برای اضافه کردن نویز در کد اومدیم یک نویز از توزیع نرمال به اندازه شکل عکس تولید کردیم. و 0.1 آن را به عکس واقعی اضافه کردیم. و عکس جدید را به عنوان ورودی به discriminator در زمان آموزش آن می‌دهیم.

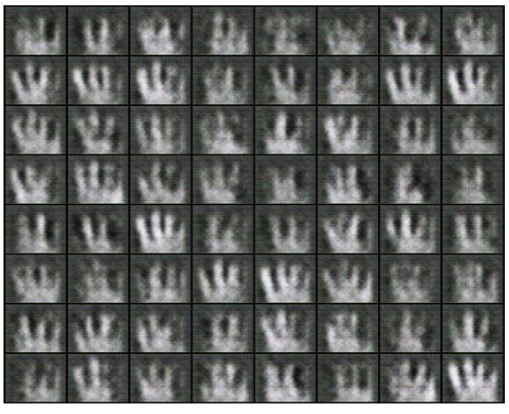
- **تصاویر تولید شده:**
- تصاویر تولید شده توسط generator در یک انیمیشن برای هر ایپاک در کد موجود می‌باشد. اینجا صرفا عکس‌های تولید شده در چند ایپاک ابتدایی و ایپاک آخر را می‌آوریم که در عکس‌های ۱۱ تا ۱۴ در ادامه آورده شده است.



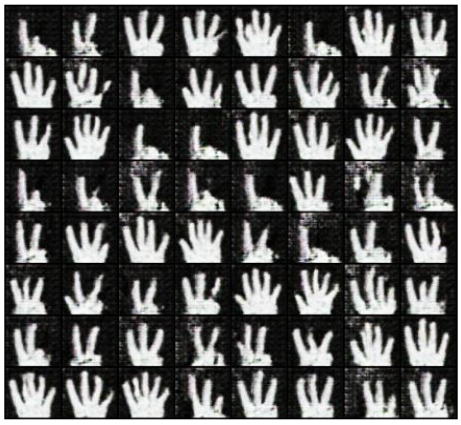
شکل ۱۱. نویز ورودی-پایدار



شکل ۱۲. تصاویر تولید شده بعد ایپاک ۱-پایدار



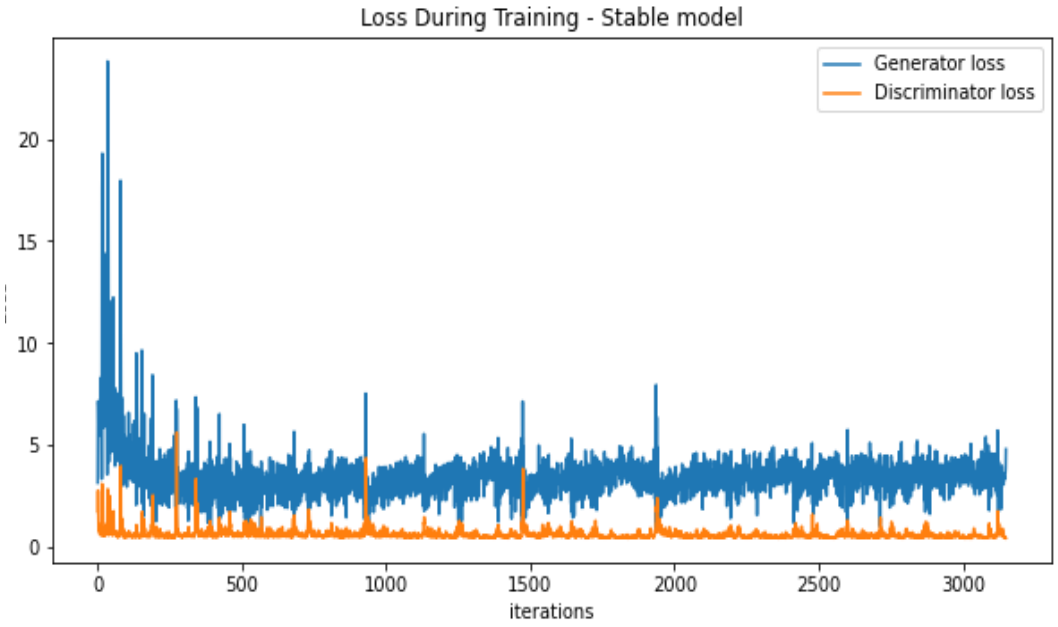
شکل ۱۳. تصاویر تولید شده بعد ایپاک ۳-پایدار



شکل ۱۴. تصاویر تولید شده بعد ایپاک ۵۰-پایدار

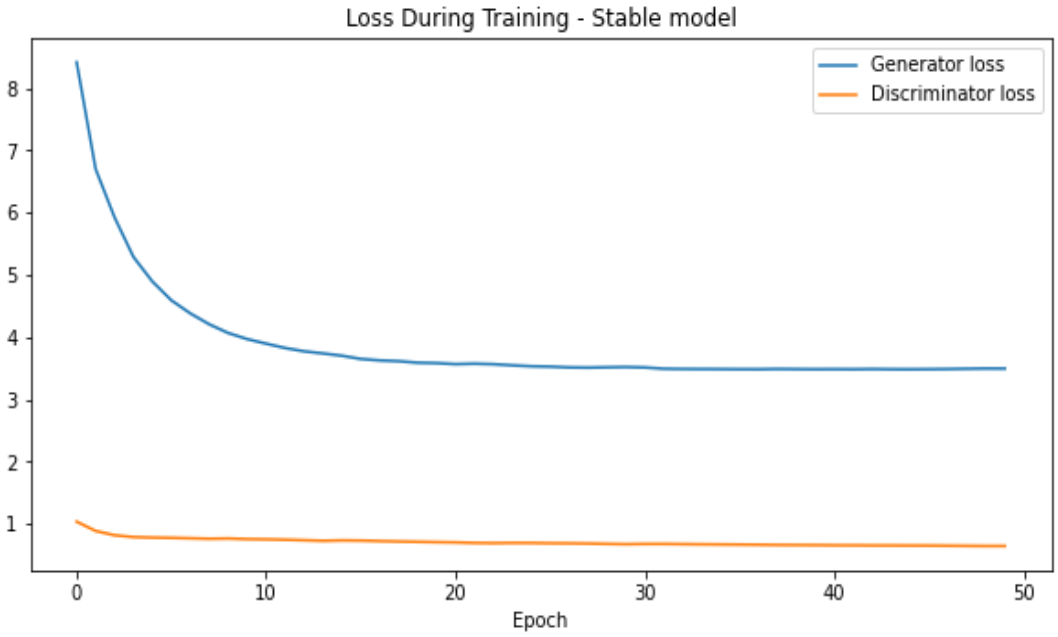
- **ارزیابی شبکه:**

نمودار loss برای هر iteration حساب شده و در شکل زیر آورده شده است:



شکل ۱۵. نمودار **loss** در طول **iterations**- مدل پایدار

نمودار loss برای هر ایپاک هم در شکل زیر آمده است که از میانگین گیری بدست آمده است:

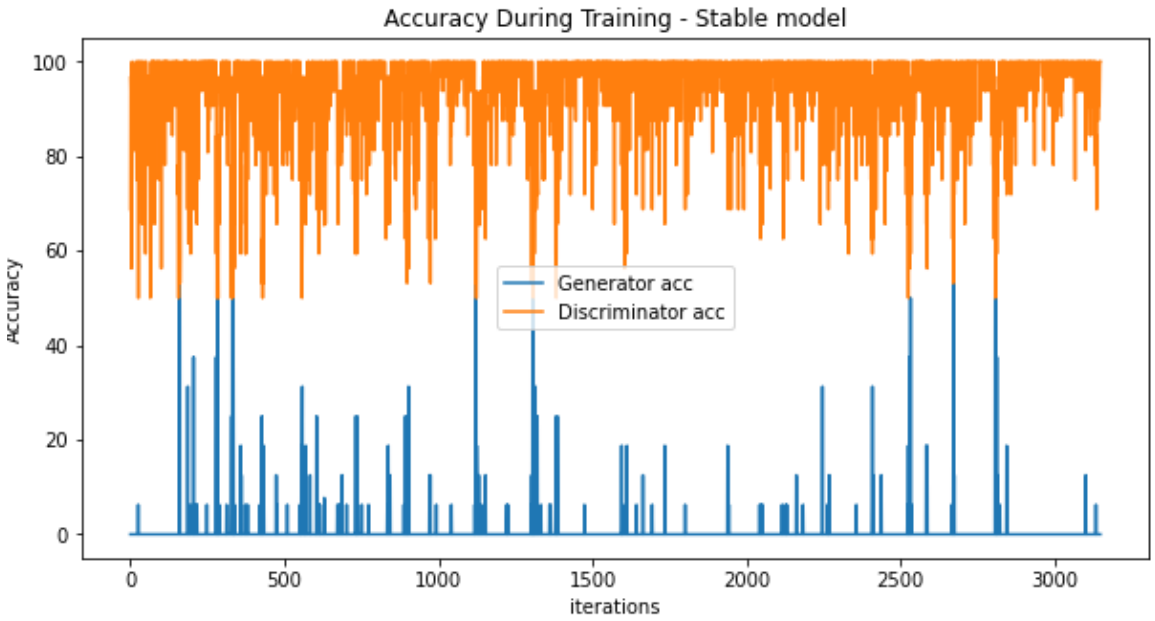


شکل ۱۶. نمودار **loss** برای هر ایپاک – مدل پایدار

اگر این نمودار را با نمودار مدل قبلی(غیرپایدار) مقایسه کنید، خواهید دید که مدل پایدر به loss نزدیک ۴ رسیده است که خیلی بهتر از loss با مقدار نزدیک 6 است. اگرچه هنوز به loss نزدیک 0 فاصله دارد. که مشکلش کماکان vanishing gradient است. که دیگر برای کاهش آن باید تابع loss را تغییر دهیم.(برای جلوگیری از پیچیده شدن و طولانی تر شدن از پیاده سازی آن‌ها صرف نظر کردیم).

اگر نویز را هم بیشتر کنیم generator نویز را هم مجبور است یاد بگیرد که باعث پایین بودن کیفیت عکس‌ها می‌شود.

نمودار accuracy:



شکل ۱۷. نمودار دقت در طول **iterations**- مدل پایدار

اینجا هم کماکان پدیده نوسانی بودن دقت را داریم. که به همان دلایل قبل است.

Wasserstein loss:

مقاله از این تکنیک استفاده نکرده اما خیلی‌ها برای بهبود مدلشان از این روش برای محاسبه loss استفاده می‌کنند که معمولا موثر هم واقع می‌شود.

توضیح مختصر درباره Wasserstein loss: ایده اش این است که به جای پیش بینی احتمال واقعی یا فیک بودن تصاویر می‌آید به واقعی بودن یا فیک بودن تصاویر نمره می‌دهد. و این نمره الزاما بین صفر و

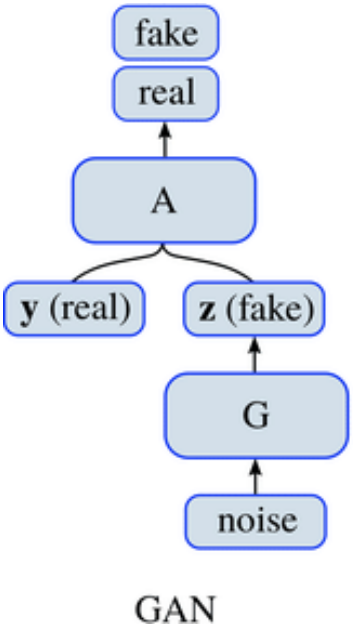
یک نیست. هدف discriminator دادن عدد بزرگتر به داده‌های واقعی نسبت به داده‌های فیک است. در واقع بیشتر از این که تمایز دهنده باشد به عنوان یک منتقد عمل می‌کند.

.

پاسخ ۲ – شبکه متخاصم مولد طبقه بند کمکی و شبکه Wasserstein

۲-۱. شبکه متخاصم مولد طبقه بند کمکی

در این بخش قصد داریم مدل AC_GAN را پیاده سازی کنیم. به طور کلی GAN یک فریمورک برای یادگیری عمیق است، که با بدست آوردن توزیع داده ورودی بتواند داده جدید با توزیع یکسان تولید کند. برای این کار از دو مدل مجزا به نام‌های generator و discriminator استفاده می‌کند. وظیفه generator تولید عکس(داده) فیک مشابه تصاویر واقعی است. وظیفه discriminator هم تشخیص دادن عکس واقعی و عکس فیک است. در زمان train، بخش generator تلاش می‌کند که با تولید عکس‌های مشابه واقعی بخش discriminator رو فریب دهد. در حالیکه بخش discriminator هم با آموزش دیدن تلاش می‌کند که داده‌های واقعی و فیک رو درست تمییز دهد.

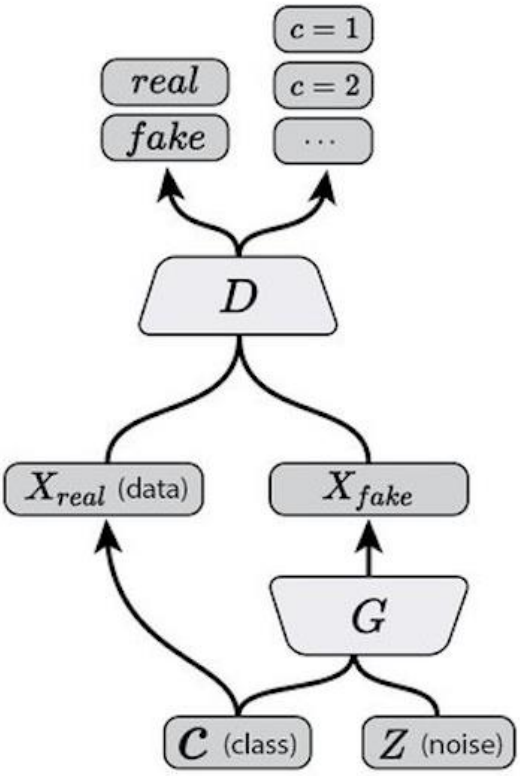


GAN

شکل ۱۸. شبکه **GAN**

در این مدل فضای latent به همراه تصاویر جعلی onehot شده سپس به وکتور نویز اضافه میشود، در نهایت آن را به شبکه generator می دهیم. این در حالیست که در AC_GAN مولد ورودی نویز دار و لیبل آن را می گیرد، خروجی آن یک عکس جعلی که به همان کلاس لیبل تعلق دارد می باشد. در این

شبکه ورودی discriminator یک عکس است و خروجی احتمال واقعی بودن آن تصویر و برچسب کلاس آن می باشد.



شکل ۱۹. شبکه **AC-GAN**

در ابتدا داده را با ماژول ImageFolder از کتابخونه torchvision پایتورچ لود میکنیم. سایز تصاویر ما 32*32 هستند. برای نتیجه گرفتن بهتر عکس‌ها را نرمالیاز هم می‌کنیم. سپس برای آن یک dataloader می‌سازیم. وزن‌های اولیه به طور تصادفی با استفاده از یک توزیع نرمال با میانگین 0 و انحراف معیار 0.2 داده شده، که این کار در تابع weights_init_normal(m) انجام گرفته است. در ادامه پارامترهای بکار رفته در مدل را خواهیم دید:

جدول ۱. **Ac-GAN hyper parameters**

Network Used Hyper Parameters	
Batch Size	64

Epochs		150
Input		32*32*3
Num classes		5
Latent Dim		105
Activation Functions	Discriminator	LeakyReLU, Sigmoid, SoftMax
	Generator	ReLU, Tanh
Loss Function	adversarial	Binary Cross Entropy
	auxiliary	Cross Entropy
Optimizer	Discriminator	Adam
	Generator	Adam
Learning rate		0.0002
Beta1		0.5
Beta2		0.999

برای پیاده سازی generator ما دارای یک لایه linear و سپس ۴ بلوک متوالی، شامل یک لایه convolutional transpose ، یک لایه Batch norm و تابع فعال ساز Relu می باشیم، تنها در بلوک آخر بعد از conv transpose از تابع فعال ساز tanh استفاده شده است. این کارها در کلاس Generator صورت گرفته است. سپس با این کلاس به راحتی یک generator می‌سازیم. و وزن‌های تصادفی اولیه را به آن اعمال می‌کنیم.

برای پیاده سازی بخش discriminator نیز معماری دارای یک بلوک کانولوشنی و سپس لایه linear می باشد. به این صورت که بلوک کانولوشنی شامل یک لایه convolutional و یک لایه batch norm است و در ادامه آن یک تابع فعال ساز LeakyRelu استفاده می‌شود. در آخر بعد از لایه کانولوشنال، دو لایه fully connected وجود دارد که برای فعال ساز از sigmoid و در لایه اخر softmax استفاده می‌شود. این کلاس را فراخوانی می‌کنیم تا یک discriminator بسازیم. و بعد از آن وزن‌های تصادفی اولیه را روی آن اعمال می‌کنیم. تا اینجا شبکه generator و discriminator آماده آموزش شده اند.

برای آموزش مدل از فرمول های زیر استفاده شده است:

$$L_S = E_x \log P(S = real|X_{real}) + E_Z \log(S = fake|X_{fake})$$
$$L_C = E_x \log P(C = c|X_{real}) + E_Z \log(C = c|X_{fake})$$

همانطور که دیده می شود، objective function دارای دو بخش می باشد: log likelihood برای correct source و همچنین log likelihood برای correct class. که discriminator تلاش دارد که $L_S + L_C$ را ماکزیمم کند در حالی که generator تلاش دارد به گونه ای آموزش ببیند که ماکزیمم مقدار $L_C - L_S$ را بدست آورد.

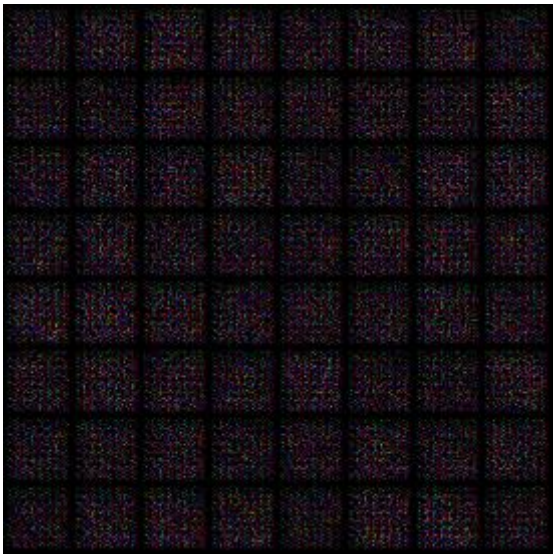
آموزش شبکه از دو قسمت تشکلیل شده است. ابتدا بخش discriminator و سپس بخش generator را آموزش میدهیم. در واقع این شبکه قابلیت تولید تصویر را باتوجه به لیبل کلاس آن را دارد. در این شبکه با دو لیبل سر و کار داریم، یکی لیبل مربوط به هر کلاس و دیگری لیبل داده واقعی و ساختگی است. لیبل تمام داده‌های ساختگی را صفر و داده‌های واقعی را یک با استفاده از دستور torch.full(batch_size, real_label) در نظر می‌گیریم.

در آموزش discriminator هدف این است که دقت تشخیص درست عکس واقعی و فیک را افزایش دهیم. داده‌ها را batch به batch آموزش می‌دهیم. به این صورت که ابتدا داده‌های واقعی را با لیبل 1 به discriminator می‌دهیم و loss را حساب کرده و سپس عملیات backward را برای پارامترهای آن انجام می‌دهیم.

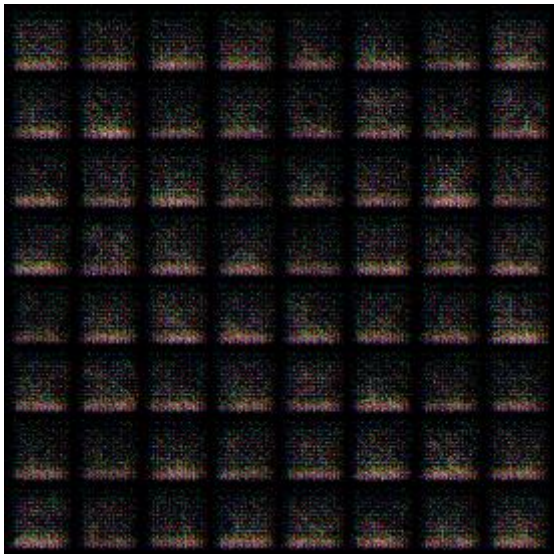
سپس از طریق generator فعلی داده فیک از نویز با استفاده از دستور torch.randn می‌سازیم. و از discriminator آن را عبور می‌دهیم. مقدار loss را حساب می‌کنیم و برای پارامترهای discriminator عملیات backward زده و وزن هایش را آپدیت می‌کنیم. توجه شود که وزن‌های generator را در این مرحله فریز می‌کنیم یعنی در backward دخالت نمی‌دهیم.

در آموزش generator هدف این است که تلاش کنیم داده فیک بهتر و نزدیک‌تر به داده واقعی بسازیم. از discriminator بخش قبل برای طبقه بندی داده تولید شده توسط generator استفاده می‌کنیم. و مقدار loss را برای generator حساب می‌کنیم. و حالا با backward مقدار وزن‌های generator را به روز می‌کنیم.

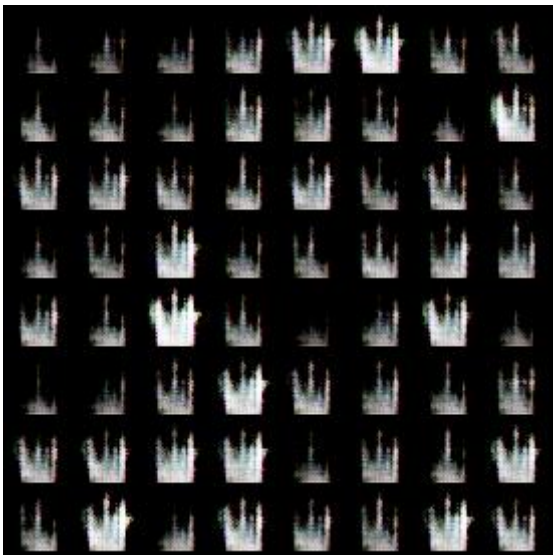
این کار را انقدر ادامه می‌دهیم که داده‌های فیک به اندازه کافی شبیه داده‌های واقعی شوند. در پایان هر ایپاک و iteration هم مقدار loss هر دو را برای گزارش کردن ذخیره می‌کنیم. همچنین بعد هر ایپاک تصاویری را توسط generator تولید کرده تا آن را ارزیابی کنیم. نتایج بدست آمده از آموزش شبکه به صورت زیر می باشد.



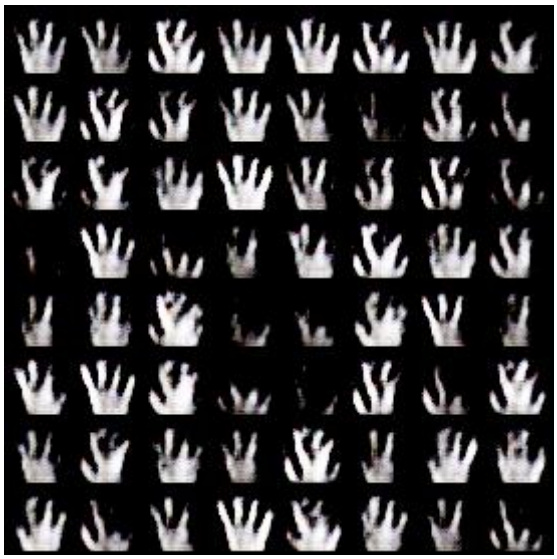
شکل ۲۰. نویز ورودی



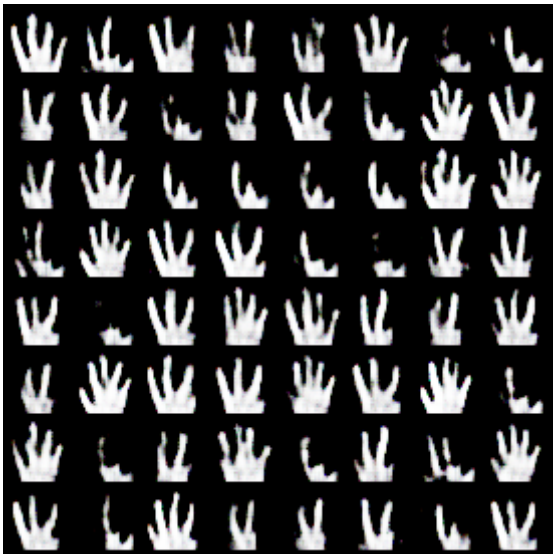
شکل ۲۱. تصاویر تولید شده بعد ایپاک ۱



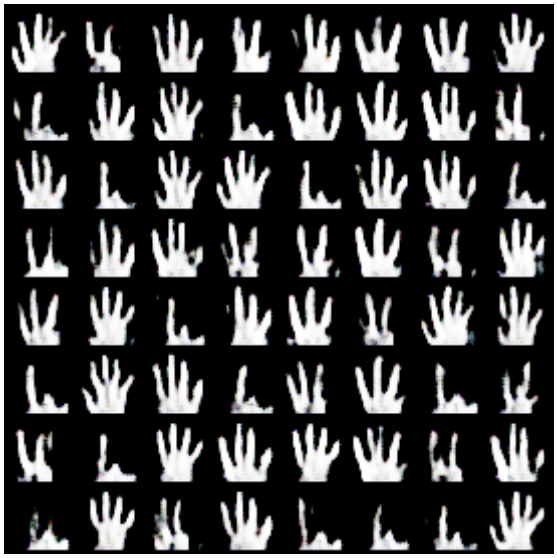
شکل ۲۲. تصاویر تولید شده بعد ایپاک ۲۰



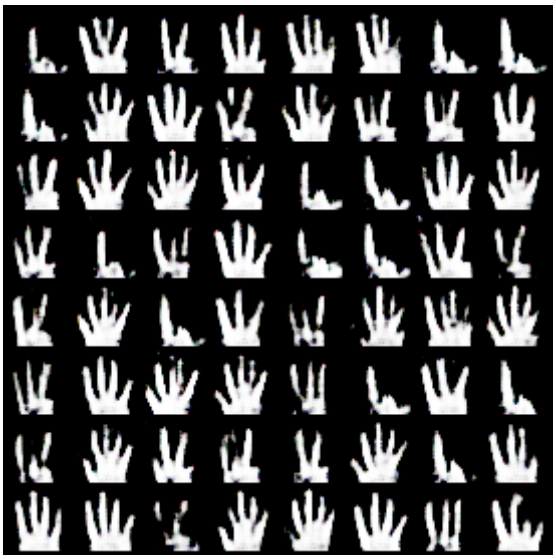
شکل ۲۳. تصاویر تولید شده بعد ایپاک ۴۰



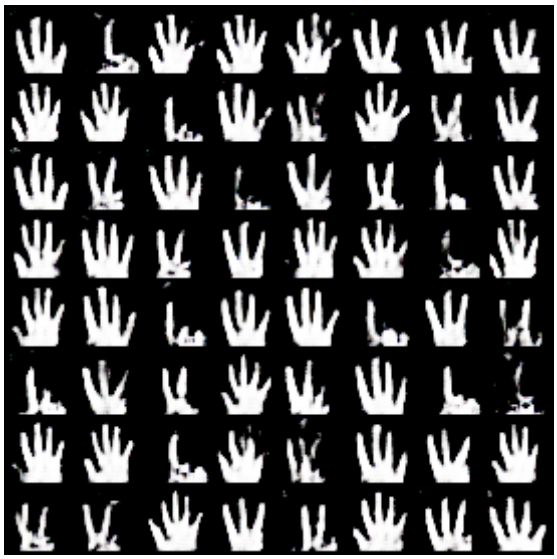
شکل ۲۴. تصاویر تولید شده بعد ایپاک ۶۰



شکل ۲۵. تصاویر تولید شده بعد ایپاک ۸۰

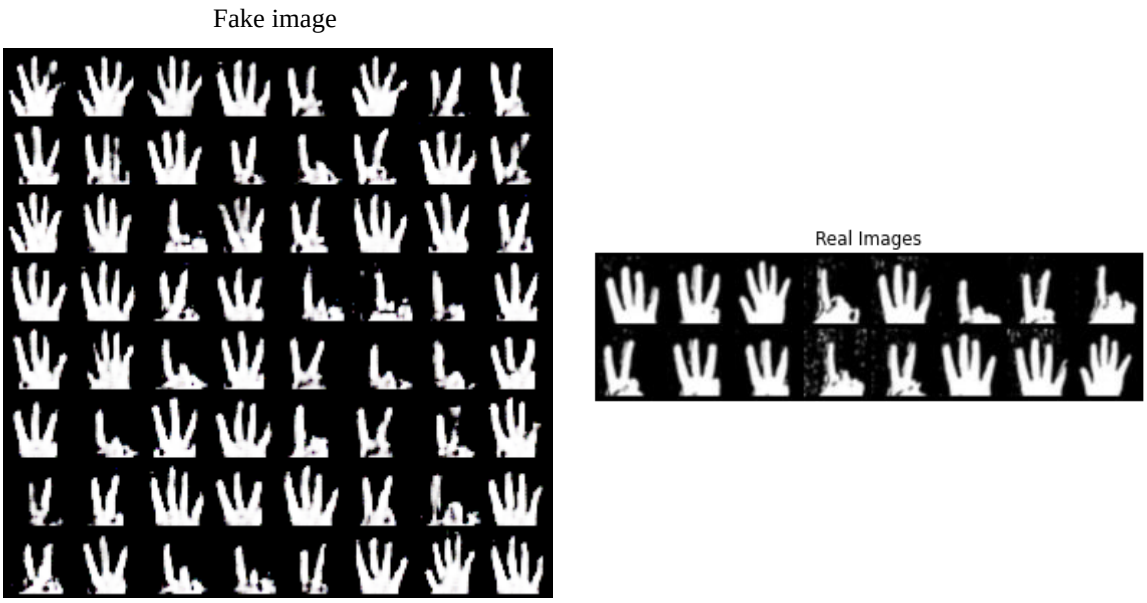


شکل ۲۶. تصاویر تولید شده بعد ایپاک ۱۰۰



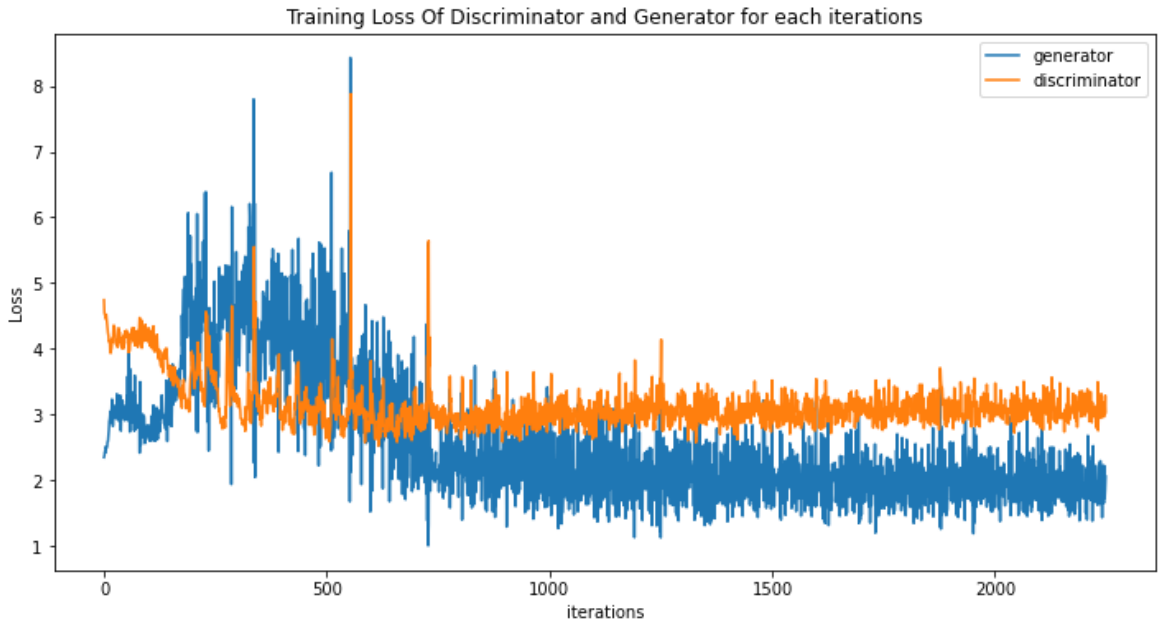
شکل ۲۷. تصاویر تولید شده بعد ایپاک ۱۲۰

همانطور که دیگه می شود مدل بعد از ۶۰ ایپاک تصاویر خوبی تولید می کند. مقایسه تصویر اصلی با تصویر ساختگی نیز نشان دهنده کارکرد خوب مدل generator می باشد.



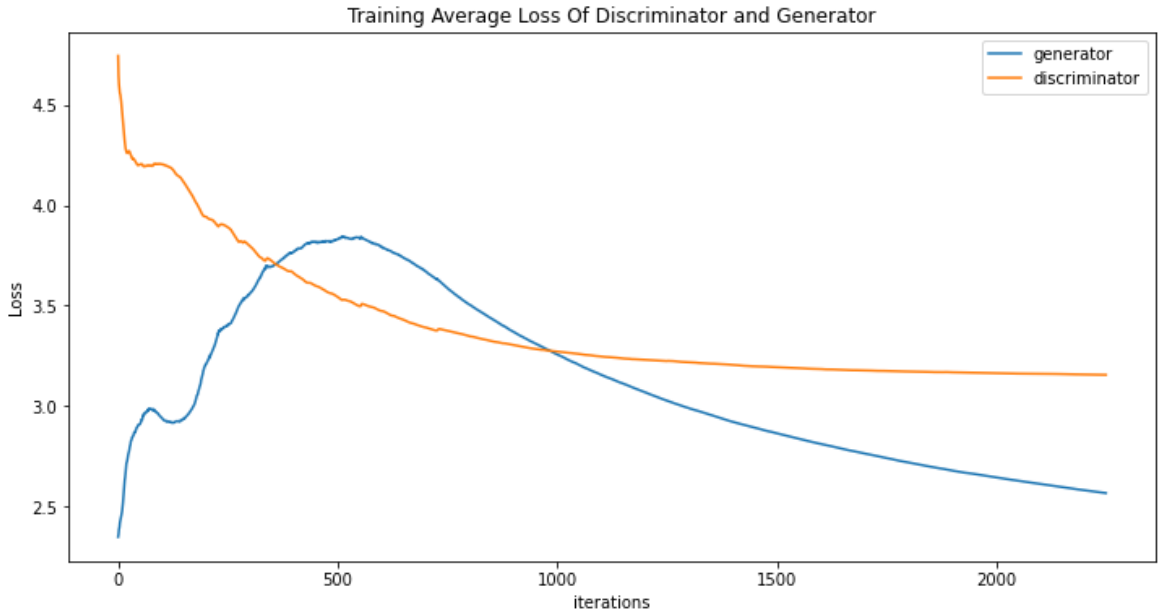
شکل ۲۸. مقایسه تصاویر واقعی و ساختگی در پایان آموزش

در ادامه نمودارهای مربوط به عملکرد شبکه را بررسی میکنیم.



شکل ۲۹. نمودار **loss** مربوط به **generator** و **discriminator** در هر **iteration**

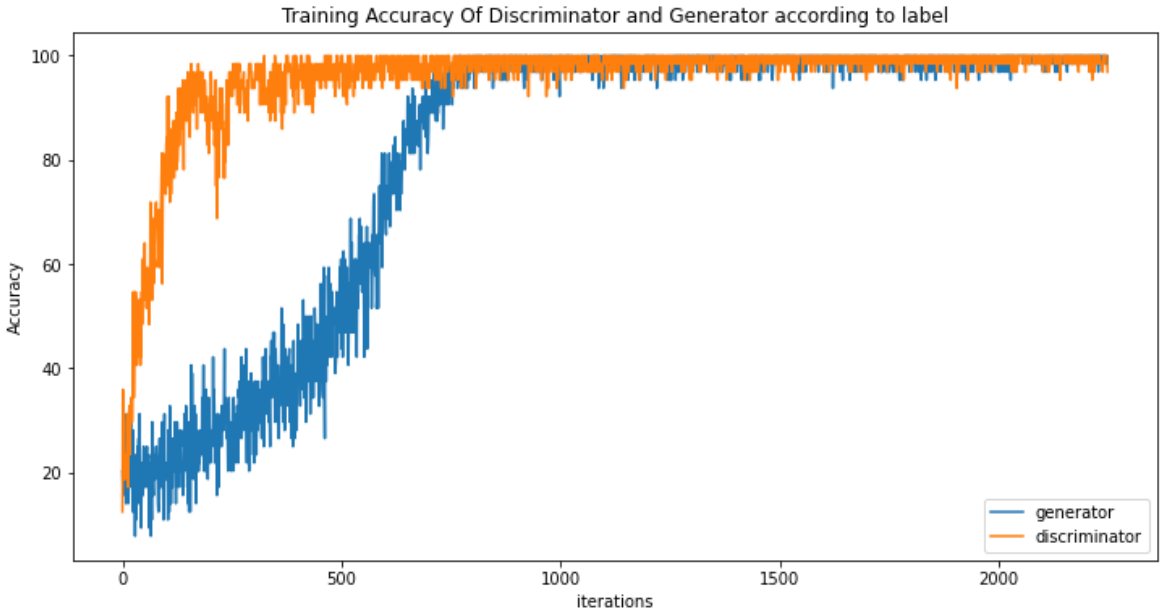
همانطور که در شکل بالا قابل مشاهده است به صورت کلی مدل generator در ابتدا یک سیر صعودی داشته ولی بعد از گذشت ۵۰۰ iteration هر دو مدل به طور کلی یک سیر نزولی را در پیش میگیرند، این موضوع در نمودار شکل ۳۰ که مربوط به loss متوسط هر ایپاک می باشد بهتر قابل رویت است.



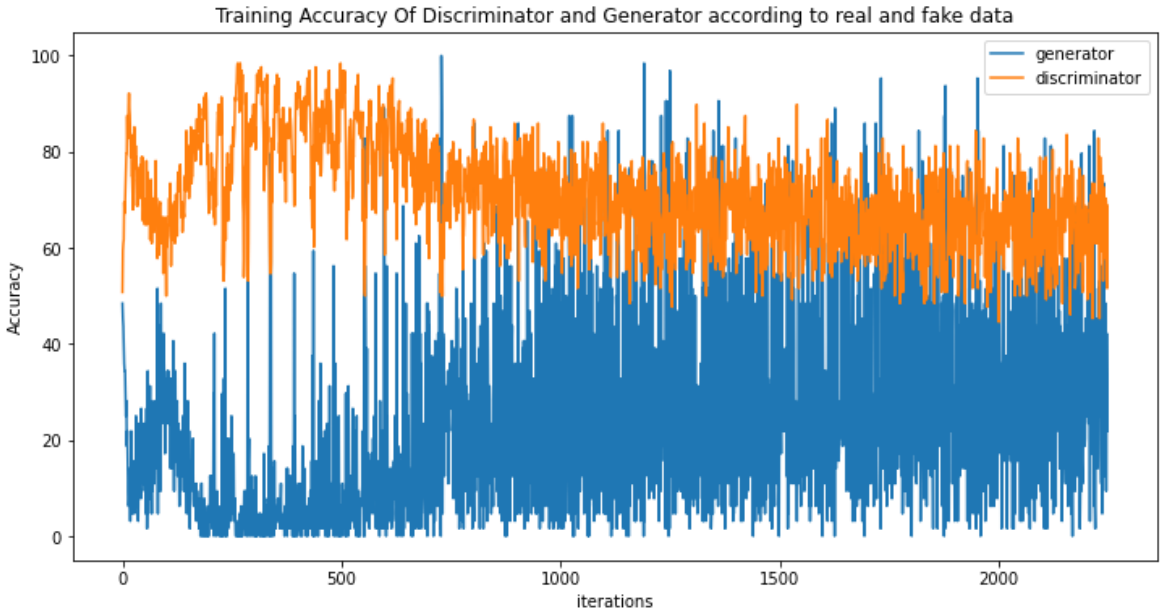
شکل ۳۰. نمودار **loss** متوسط در هر ایپاک مربوط به **generator** و **discriminator**

با توجه به نمودار شکل ۲۹ و ۳۰ می توان گفت، مطابق با آنچه انتظار می رفت، مدل generator در ابتدا خوب عمل نمیکند و تصاویر تولید توسط آن قابل شناسایی است ولی بعد از گذشت تقریبا ۵۰۰ iteration عملکرد آن بهبود یافته و loss آن سیر نزولی گرفته، یعنی تصاویری که تولید کرده کمتر قابل شناسایی است و بیشتر تصویر واقعی در نظر گرفته شده است.

در ادامه دقت مدل را به دو روش بررسی میکنیم، در واقع ما دو نوع دقت برای مدل بدست آورده ایم. یکی دقت مدل با استفاده از بررسی تصاویر تولید شده و لیبل اختصاص داده شده به آن تصاویر می باشد، و روش دیگر به این صورت است اگر داده های واقعی و همچنین داده های ساختگی (تولید شده توسط generator) را به dicriminator بدهیم و احتمال اختصاص داده شده به یک تصویر تولید شده توسط generator کمتر از ۵۰ درصد باشد آن را ساختگی و همچنین اگر احتمال داده شده به یک تصویر واقعی بیشتر از ۵۰درصد باشد آن را واقعی در نظر میگیریم و جمع آن ها را حساب کرده و دقت را محاسبه می کنیم. برای generator نیز به همین شکل عمل میکنیم، به این صورت که اگر تصاویر تولید شده توسط generator را به مدل discriminator بدهیم و خروجی احتمال بالاتر از ۵۰ درصد بدهد آن تصویر، واقعی در نظر گرفته می شود و مجموع آنها را محاسبه کرده سپس دقت generator را محاسبه میکنیم.



شکل ۳۱. دقت مدل **generator** و **discriminator** با استفاده از روش اول - با کمک **label**



شکل ۳۲. دقت مدل **generator** و **discriminator** با استفاده از روش دوم - با کمک احتمال خروجی

.

۲-۲. شبکه متخاصم مولد Wasserstein

در این بخش قصد داریم شبکه DCgan را با کمک Wasserstein loss پیاده سازی کنیم. در واقع از روش Wasserstein برای بهبود مدل برای محاسبه loss و همچنین بهبود مشکل vanishing gradient و mode-collapse استفاده می‌کنند که معمولا موثر هم واقع می‌شود. در حقیقت Wasserstein GAN هم باعث بهبود stability زمان آموزش مدل می شود و بعلاوه یم تابع loss که در ارتباط با کیفیت تصاویر تولید شده می باشد ایجاد میکند.

ایده Wasserstein loss این است که به جای پیش بینی احتمال واقعی یا جعلی بودن تصاویر سعی میکند به واقعی بودن یا جعلی بودن تصاویر نمره دهد. و این نمره الزاما بین صفر و یک نیست. هدف discriminator دادن عدد بزرگتر به داده‌های واقعی نسبت به داده‌های جعلی است. در واقع بیشتر از این که تمایز دهنده باشد به عنوان یک منتقد عمل می‌کند.

در واقع Wasserstein loss تابع loss مربوط به GAN را به صورتی فرموله میکند که مینیمم فاصله بین احتمال توزیع ها را واضح تر نمایش دهد. و همچنین این loss برای رفع مشکل ناشی از تابع اصلی loss مربوط به GAN به عنوان بازی zero-sum (minimax) طراحی شده است. در این روش weight clipping زمانی که داده پیچیده داشته باشیم نتیجه خوب بدست نمی آورد ولی با داده ساده می تواند خروجی مطلوبی داشته باشد.

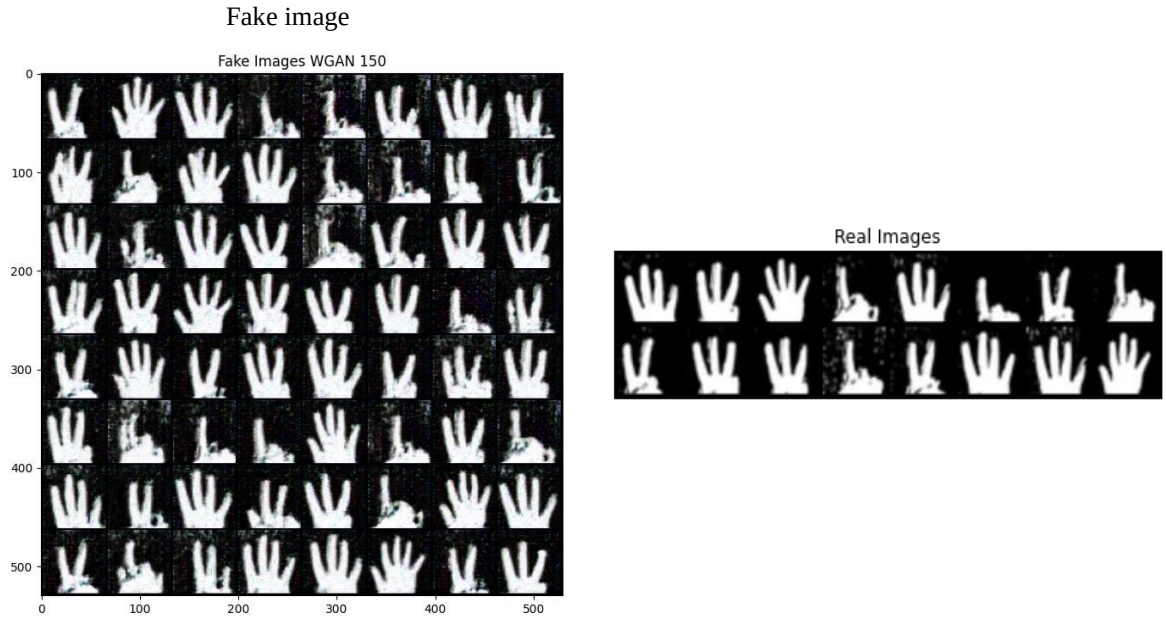
برای بدست آوردن این loss ابتدا میانگین خروجی discriminator با داده های واقعی و همچنین داده های جعلی را بدست آورده و سپس میانگین خروجی مربوط به داده های جعلی را منهی میانگین خروجی مربوط به داده های واقعی میکنیم.

جدول ۲. **WGAN hyper parameters**

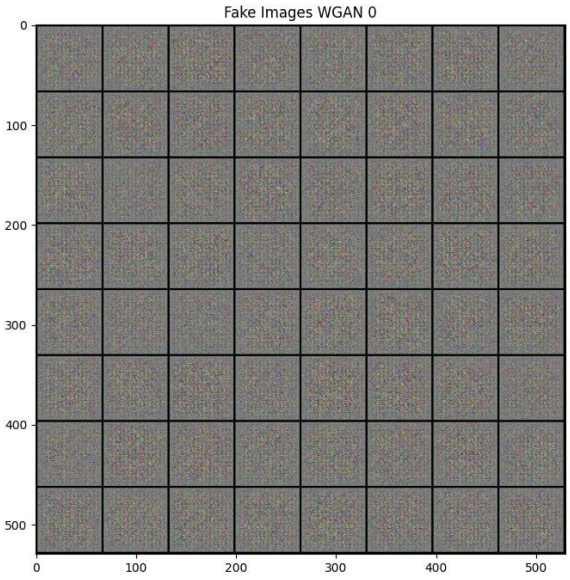
Network Used Hyper Parameters		
Batch Size		16
Epochs		150
Input		64*64*3
Num classes		5
Latent Dim		100
Activation Functions	Discriminator	LeakyReLU, Sigmoid

	Generator	ReLU, Tanh
Loss Function		Wasserstein loss (weight clipping)
Optimizer	Discriminator	RMSprop
	Generator	RMSprop
Learning rate		0.00025

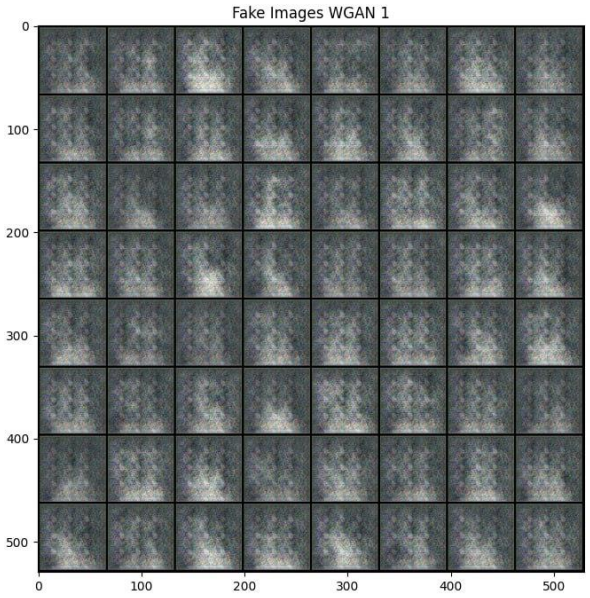
با استفاده از پارامترهای بالا مدل را همانند آنچه پیشتر در بخش ۱ سوال اول گفته شد پیاده سازی کرده با این تفاوت که از تابع خطای Wasserstein loss و تابع optimizer، Root Mean Squared Propagation برای مدل generator و discriminator استفاده شده است. نتایج بدست آمده از این مدل Wgan در ادامه آورده شده است:



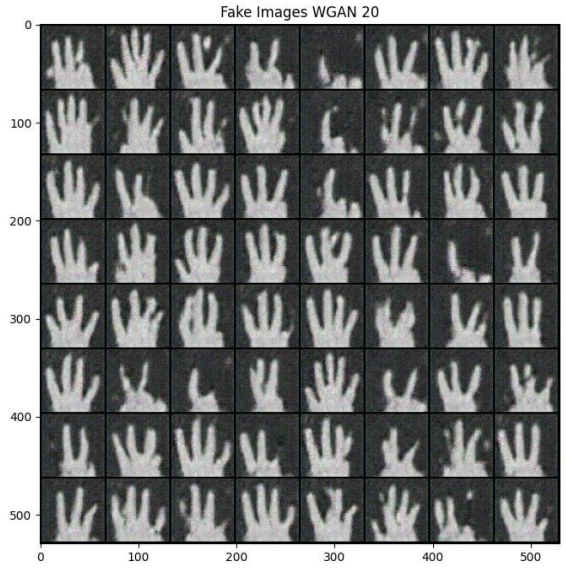
شکل ۳۳. مقایسه تصاویر واقعی و ساختگی در پایان آموزش



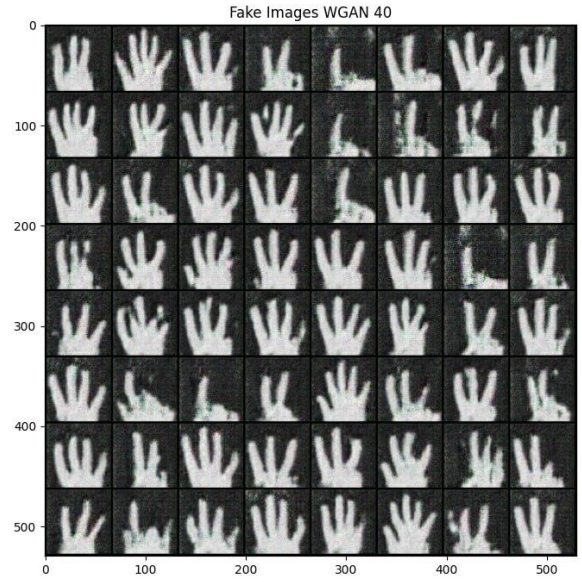
شکل ۳۴. نویز ورودی



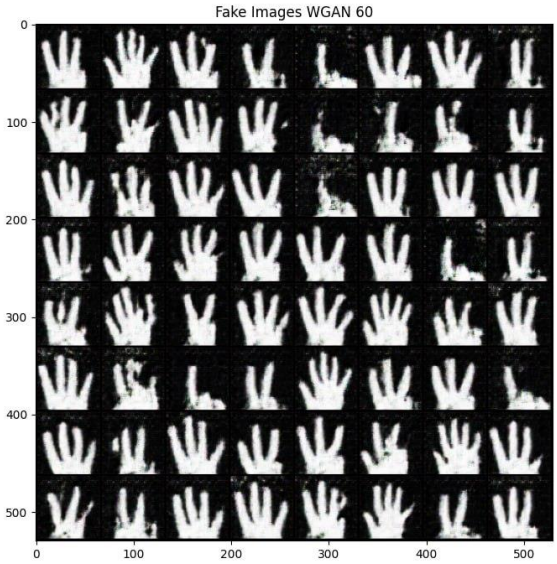
شکل ۳۵. تصاویر تولید شده بعد ایپاک ۱



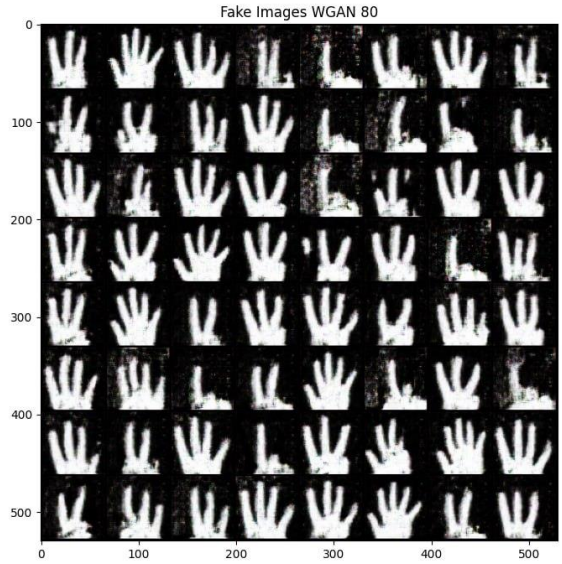
شکل ۳۶. شکل ۳۵. تصاویر تولید شده بعد ایپاک ۲۰



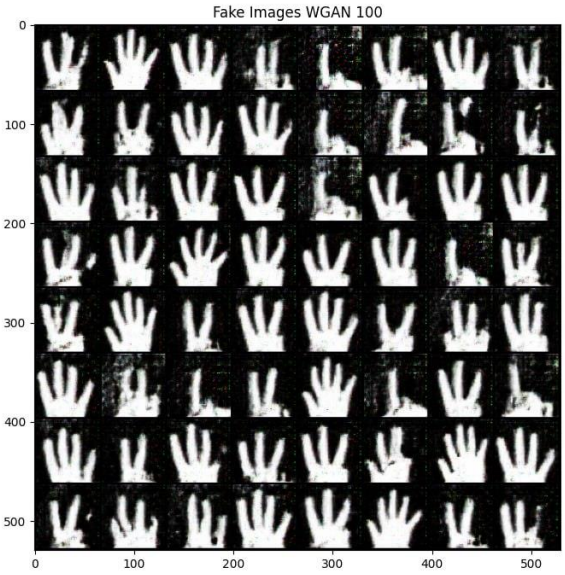
شکل ۳۷. تصاویر تولید شده بعد ایپاک ۴۰



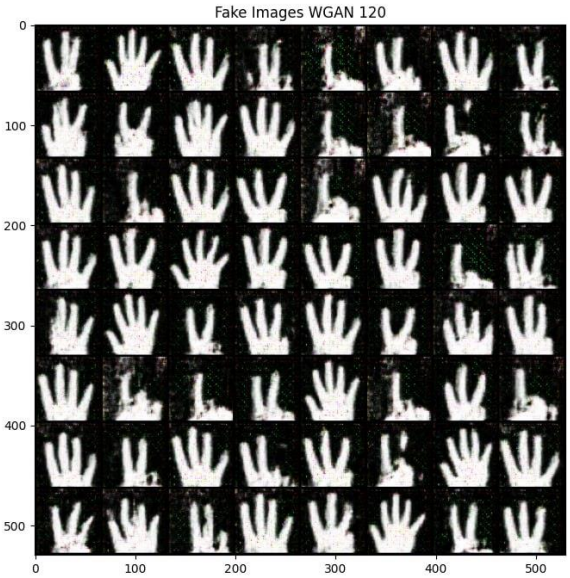
شکل ۳۸. تصاویر تولید شده بعد ایپاک ۶۰



شکل ۳۹. تصاویر تولید شده بعد ایپاک ۸۰



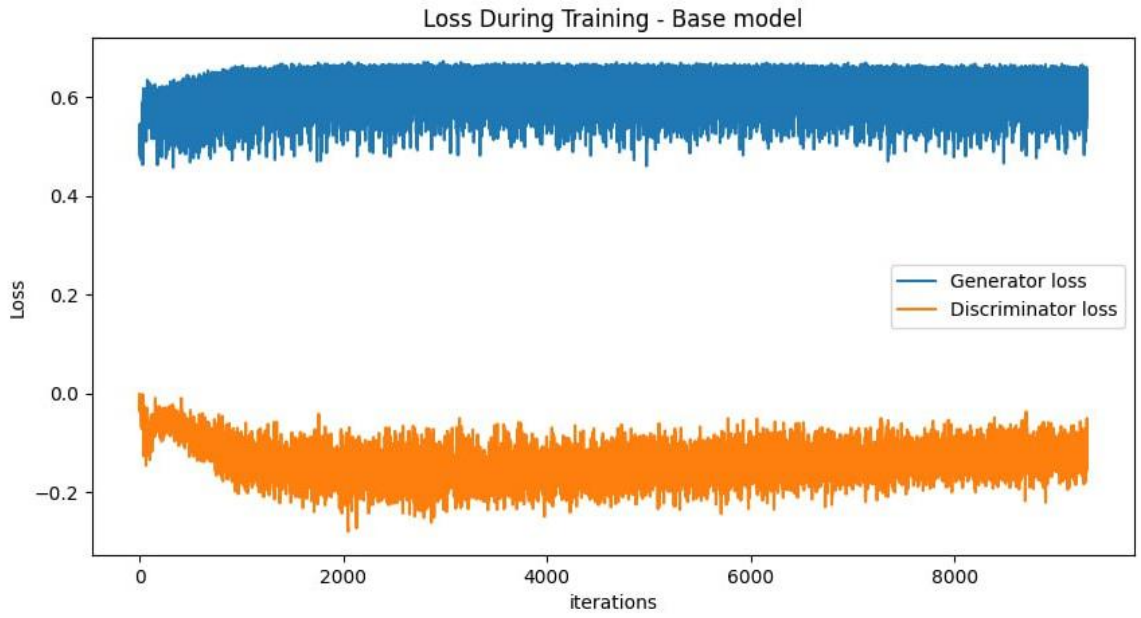
شکل ۴۰. تصاویر تولید شده بعد ایپاک ۱۰۰



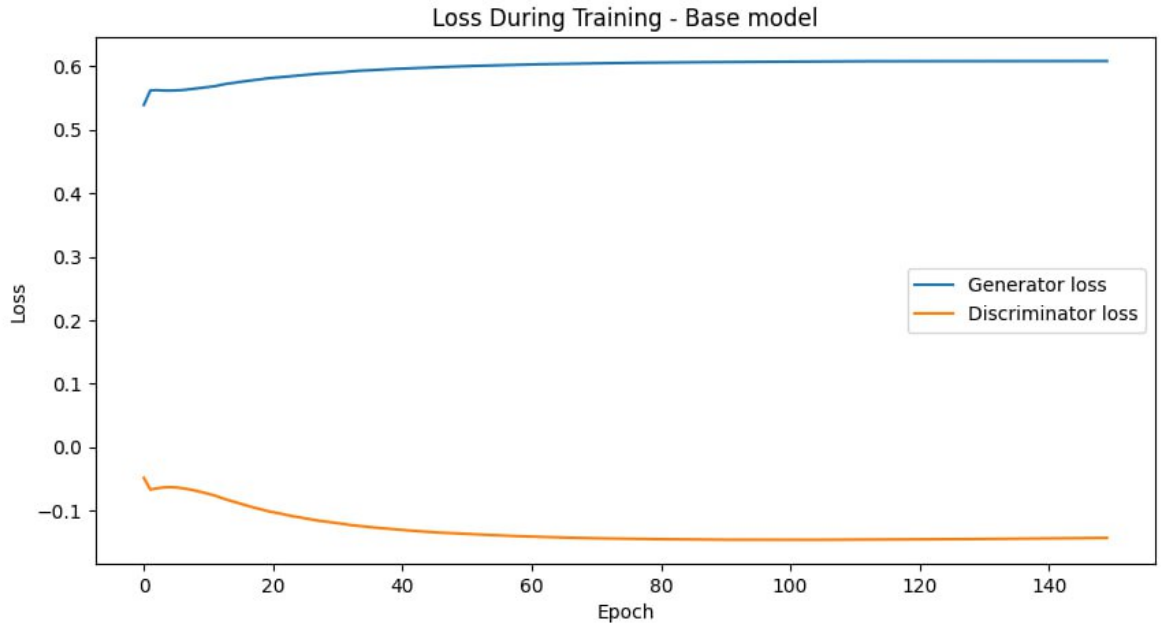
شکل ۴۱. تصاویر تولید شده بعد ایپاک ۱۲۰

همانطور که دیگه می شود مدل بعد از ۶۰ ایپاک تصاویر خوبی تولید می کند. مقایسه تصویر اصلی با تصویر جعلی نیز نشان دهنده کارکرد خوب مدل generator می باشد.

در ادامه نمودارهای مربوط به عملکرد شبکه را بررسی میکنیم:



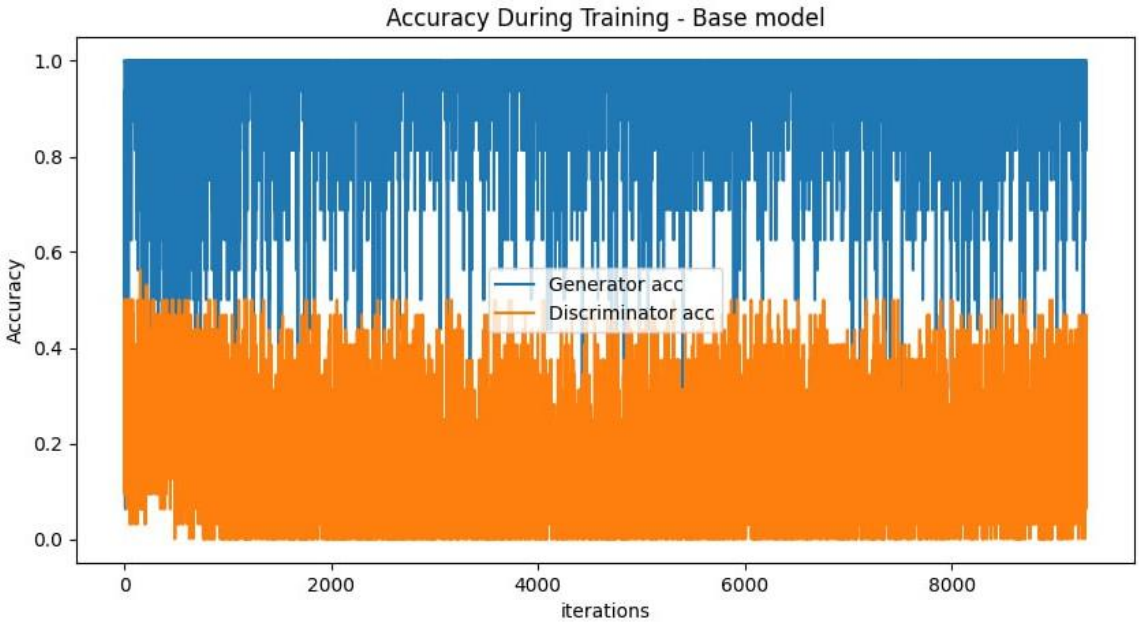
شکل ۴۲. نمودار **loss** مربوط به **generator** و **discriminator** در هر **iteration**



شکل ۴۳. نمودار **loss** متوسط در هر اپیاک مربوط به **generator** و **discriminator**

اگر این نمودار را با نمودار مدل قبل(dcgan) مقایسه کنیم، خواهیم دید که در این مدل خطا بسیار کاهش یافته و به loss نزدیک 0 رسیده است و مشکل vanishing gradient را برطرف ساخته است.

با توجه به اینکه Wasserstein loss (weight clipping) به مدل discriminator کمک میکند تا عملکرد بهتری داشته باشد بنابراین نوسانات مربوط به مدل کمتر شده است و سریع تر یاد گرفته است.



شکل ۴۴. دقت مدل **generator** و **discriminator**

دقت مربوط به generator تقریبا بین بازه ۵۰ تا ۱۰۰ درصد متغیر می باشد که طبع آن دقت مدل discriminator در بازه ۰ تا ۵۰ درصد قرار میگیرد.