



به نام خدا



دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

مینی‌پروژه سری ۳

نام و نام خانوادگی	مهدی عبدالله چالکی - مهدی مهدیخانی
شماره دانشجویی	۸۱۰۶۰۰۲۹۷-۸۱۰۶۰۰۲۹۰
تاریخ ارسال گزارش	۱۴۰۱ تیر

فهرست گزارش سوالات

۱ سوال ۱ – SGAN
۱ الف) توضیح مختصر از نحوه‌ی آموزش شبکه
۷ ب) آموزش شبکه‌ی Semi-Supervised GAN
۱۰ ج) آموزش شبکه Classifier و تشکیل جدول مقایسه
۲۰ د) استفاده از Variational autoencoder به عنوان Generator
۳۱ سوال ۲ – DCGAN
۳۱ الف)
۳۲ ب)
۳۹ ج)
۳۹ مشکل عدم‌همگرایی
۳۹ مشکل محو شدن گرادیان و mode collapse
۴۰ استفاده از Wasserstein loss
۴۳ رفع مشکل همگرایی
۴۵ رفع مشکل (WGAN-GP) WGAN
۴۹ سوال ۳ – VQ-VAE
۴۹ الف)
۵۰ ب)
۵۱ ج)
۶۰ د)
۶۵ مراجع

سوال ۱ – SGAN

در این سوال قصد داریم با کمک یک مدل تغییر یافته از شبکه‌ی GAN، به یادگیری نیمه ناظارتی (Semi-supervised) بپردازیم. در تنظیمات کلاس‌بندی نیمه ناظارتی، ما تعداد کمی داده‌ی برچسب‌دار و GAN تعداد زیادی داده‌ی بدون برچسب داریم. در این سوال می‌خواهیم از قابلیت آموزش بدون ناظارت GAN برای کمک به کلاسیفایر در تنظیمات نیمه ناظارتی استفاده کنیم.

در مقاله‌ای با عنوان "Semi-Supervised Learning with Generative Adversarial Networks" این روش معرفی شده‌است. در این مقاله، بخش discriminator شبکه‌های GAN معمولی تغییر داده شده‌است و علاوه بر خروجی ۰ یا ۱ (برای تشخیص واقعی یا جعلی بودن داده‌ها)، به تعداد کلاس‌های داده‌آموزشی (N) نیز خروجی دارد. یعنی در مجموع $N+1$ خروجی داریم که N تا از آن برای تشخیص احتمال وقوع هر کلاس، و یکی برای تشخیص احتمال جعلی بودن داده مورد استفاده قرار می‌گیرد. استفاده از این روش باعث می‌شود تا کلسیفایر دارای بازدهی بهتری باشد و نیز شبکه GAN بتواند تصاویر بهتری را تولید کند.

ایده کلی در این مقاله آن است که بهبود بخش discriminator (که آن را D می‌نامیم) باعث بهبود عملکرد بخش classifier (که آن را C می‌نامیم) می‌شود، و همچنین بهبود C باعث بهبود عملکرد D می‌شود (که می‌دانیم باعث بهتر شدن بخش جنریتور (G) می‌شود). بنابراین می‌توان با ایجاد یک حلقه فیدبک، بتوان کاری کرد که این سه بخش به یکدیگر کمک کنند تا عملکرد هر کدام بهتر شود.

الف) توضیح مختصر از نحوه آموزش شبکه

برای آموزش این شبکه، لازم است تا ابتدا دو بخش مجزای شبکه، یعنی بخش D و G تعریف شوند.

بخش :Discriminator/Classifier

نحوه کلی کار این بخش بدین صورت است که داده‌هایی را در ابعاد $28*28$ دریافت کرده، و در خروجی اولاً مشخص کند که این تصویر واقعی است یا جعلی، و همچنین بتواند لیبل آن تصویر را مشخص کند. برای ایجاد این ساختار، روش‌های مختلفی وجود دارد. در یک روش، می‌توان دو شبکه تعریف کرد که وزن‌های مشترکی دارند. در این حالت، آپدیت کردن یکی از شبکه‌ها، آن یکی را نیز آپدیت می‌کند. همچنین این نوع شبکه دارای دو لایه‌ی خروجی است که بخش D دارای یک نورون خروجی، و بخش C دارای ۱۰ نورون خروجی است.

در حالت دوم، می‌توان شبکه را به صورتی تعریف کرد که دارای دو خروجی باشد. عیب این روش آن است که تصاویر فیک هم می‌توانند وارد شبکه شوند و بنابراین بخش C باید یک خروجی اضافه نیز داشته باشد که مشخص کند تصویر از انواع دهگانه نیست. (این مدل برخلاف توضیحات مقاله است).

در حالت آخر که در این تمرین استفاده شده است، استفاده از مدل stacked است که خروجی لایه‌ی قبل از فعال‌ساز مدل supervised برداشته شده، جمع نمایی آن‌ها حساب می‌شود و سپس نرمالایز می‌شود. این کار کمک می‌کند تا بدون آنکه نیاز به لیبل اضافی یا محاسبه از ابتدا برای هر شبکه باشد، بتوانیم در یک مرحله فیدفوروارد، خروجی مناسب برای دو بخش C و D چاپ شود. از این روش در مقاله‌ای تحت عنوان "Improved Techniques for Training GAN" استفاده شده است. در نهایت یک لایه‌ی فعال‌ساز softmax این خروجی‌ها را نرمالایز کرده و به عنوان خروجی بخش D می‌دهد. در نتیجه برای تصاویر واقعی، خروجی نزدیک به ۱ و برای تصاویر جعلی، خروجی نزدیک به ۰ خواهد بود. ساختار مدل بخش D به شکل زیر است:

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[None, 28, 28, 1]	0
conv2d (Conv2D)	(None, 14, 14, 128)	1280
leaky_re_lu (LeakyReLU)	(None, 14, 14, 128)	0
conv2d_1 (Conv2D)	(None, 7, 7, 128)	147584
leaky_re_lu_1 (LeakyReLU)	(None, 7, 7, 128)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	147584
leaky_re_lu_2 (LeakyReLU)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dropout (Dropout)	(None, 2048)	0
dense (Dense)	(None, 10)	20490
lambda (Lambda)	(None, 1)	0
=====		
Total params:	316,938	
Trainable params:	316,938	
Non-trainable params:	0	

شکل ۱: ساختار بخش Discriminator

که لایه‌ی lambda، تابع فعال‌سازی که پیش‌تر توضیح داده شد را اعمال می‌کند. این تابع به شکل زیر است:

$$D(x) = \frac{Z(x)}{Z(x) + 1}$$

$$Z(x) = \sum_{k=1}^K \exp [l_k(x)]$$

همچنین ساختار بخش C به شکل زیر خواهد بود:

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d (Conv2D)	(None, 14, 14, 128)	1280
leaky_re_lu (LeakyReLU)	(None, 14, 14, 128)	0
conv2d_1 (Conv2D)	(None, 7, 7, 128)	147584
leaky_re_lu_1 (LeakyReLU)	(None, 7, 7, 128)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	147584
leaky_re_lu_2 (LeakyReLU)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dropout (Dropout)	(None, 2048)	0
dense (Dense)	(None, 10)	20490
activation (Activation)	(None, 10)	0
<hr/>		
Total params: 316,938		
Trainable params: 316,938		
Non-trainable params: 0		

شکل ۲: ساختار بخش Classifier

در این شبکه‌ها، ورودی تصویر که سایز ۲۸*۲۸ دارد وارد می‌شود و با استفاده از لایه‌های کانولوشنی توابع فعال‌ساز Leaky ReLU که نمونه بهبود یافته ReLU است، به ابعاد ۴*۴ با ۱۲۸ فیلتر تبدیل می‌شود و سپس با لایه‌های dense و flatten، به مقادیر نهایی خود می‌رسند.

بخش :Generator

در این قسمت، باید یک جنریتور تعریف شود. ورودی این جنریتور تعدادی داده از فضای latent است که از توزیعی رندوم سمپل شده‌اند. سپس این بردار داده‌ها رندوم توسعه تعدادی لایه‌ی ترانهاده کانولوشنی، به ابعاد بالاتری می‌رسند تا در نهایت خروجی آن به شکل تصاویر عادی (۲۸*۲۸) دریابید. معماری این شبکه به شکل زیر است:

Layer (type)	Output Shape	Param #
=====		
input_2 (InputLayer)	[None, 100]	0
dense_1 (Dense)	(None, 6272)	633472
leaky_re_lu_3 (LeakyReLU)	(None, 6272)	0
reshape (Reshape)	(None, 7, 7, 128)	0
conv2d_transpose (Conv2DTra nspose)	(None, 14, 14, 128)	262272
leaky_re_lu_4 (LeakyReLU)	(None, 14, 14, 128)	0
conv2d_transpose_1 (Conv2DT ranspose)	(None, 28, 28, 128)	262272
leaky_re_lu_5 (LeakyReLU)	(None, 28, 28, 128)	0
conv2d_3 (Conv2D)	(None, 28, 28, 1)	6273
=====		
Total params: 1,164,289		
Trainable params: 1,164,289		
Non-trainable params: 0		

شکل ۳: ساختار بخش Generator

شبکه‌ی GAN

پس از تعریف دو بخش C/D و G، می‌توانیم از اتصال این شبکه‌ها به یکدیگر، شبکه‌ی GAN نهایی را تولید کنیم. در این شبکه که برای آموزش بخش G مورد استفاده قرار می‌گیرد، وزن‌های C/D فریز شده و تنها بخش G آموزش می‌بیند. خطا‌ی آن از نوع Binary Crossentropy بوده و با بهینه‌ساز Adam آموزش می‌بیند. ساختار این شبکه به صورت زیر است:

Layer (type)	Output Shape	Param #
=====		
input_2 (InputLayer)	[(None, 100)]	0
dense_1 (Dense)	(None, 6272)	633472
leaky_re_lu_3 (LeakyReLU)	(None, 6272)	0
reshape (Reshape)	(None, 7, 7, 128)	0
conv2d_transpose (Conv2DTra nspose)	(None, 14, 14, 128)	262272
leaky_re_lu_4 (LeakyReLU)	(None, 14, 14, 128)	0
conv2d_transpose_1 (Conv2DT ranspose)	(None, 28, 28, 128)	262272
leaky_re_lu_5 (LeakyReLU)	(None, 28, 28, 128)	0
conv2d_3 (Conv2D)	(None, 28, 28, 1)	6273
model_1 (Functional)	(None, 1)	316938
=====		
Total params:	1,481,227	
Trainable params:	1,164,289	
Non-trainable params:	316,938	

شکل ۴: ساختار بخش GAN

تولید داده‌های مورد نیاز:

در این بخش، لازم است تا داده‌های مورد نیاز برای آموزش شبکه تولید شوند. دیتای آموزشی مورد نیاز، دیتاست MNIST است که شامل تصاویر دستنویس ارقام ۰ تا ۹ است. در بخش `load_real_samples` این دیتاست لود شده، ابعاد آن به صورت $28 \times 28 \times 1$ در می‌آید و سپس عملیات نرمالایز کردن بر روی آن انجام می‌شود.

پس از لود دیتاست، باید تصاویری که برای آموزش در حالات `unsupervised` و `supervised` استفاده می‌شوند، انتخاب شوند. در تابع `select_supervised_samples` به تعداد کلاس‌های موجود (در اینجا ۱۰ عدد برای ارقام ۰ تا ۹) و متناسب با تعداد سمپل‌های خواسته شده (در این تمرین ۱۰۰، ۵۰ و ۲۵) داده به همراه لیبل آن‌ها جدا می‌شود. در بخش بعدی با کمک تابع `generate_real_samples`، داده‌های واقعی (برای آموزش `real`) از دیتاست اصلی انتخاب می‌شود. این داده‌ها دارای لیبل ۱ هستند که برای تست کردن عملکرد شبکه در تشخیص جعلی یا واقعی بودن تصاویر مورد استفاده قرار می‌گیرند.

در قسمت بعدی، لازم است تا داده‌های جعلی برای آموزش شبکه تولید شود. بدین منظور، ابتدا باید تعدادی داده به ابعاد فضای `latent` (در اینجا ۱۰۰) به صورتی که کورلیشن نداشته باشند (یعنی داده‌ها مستقل از هم باشند – بردار نهایی به صورت نویز باشد) سمپل شوند. این کار توسط تابع `generate_latent_points` انجام می‌گیرد. سپس با استفاده از این داده‌ها و به کمک بخش `G` شبکه، ابعاد ورودی آپ‌سمپل شده تا به خروجی برسد. در نتیجه این عمل که در تابع `generate_fake_samples` اتفاق می‌افتد، داده‌هایی جعلی در ابعاد تصاویر اصلی خواهیم داشت که می‌تواند وارد `C/D` شود.

آموزش شبکه

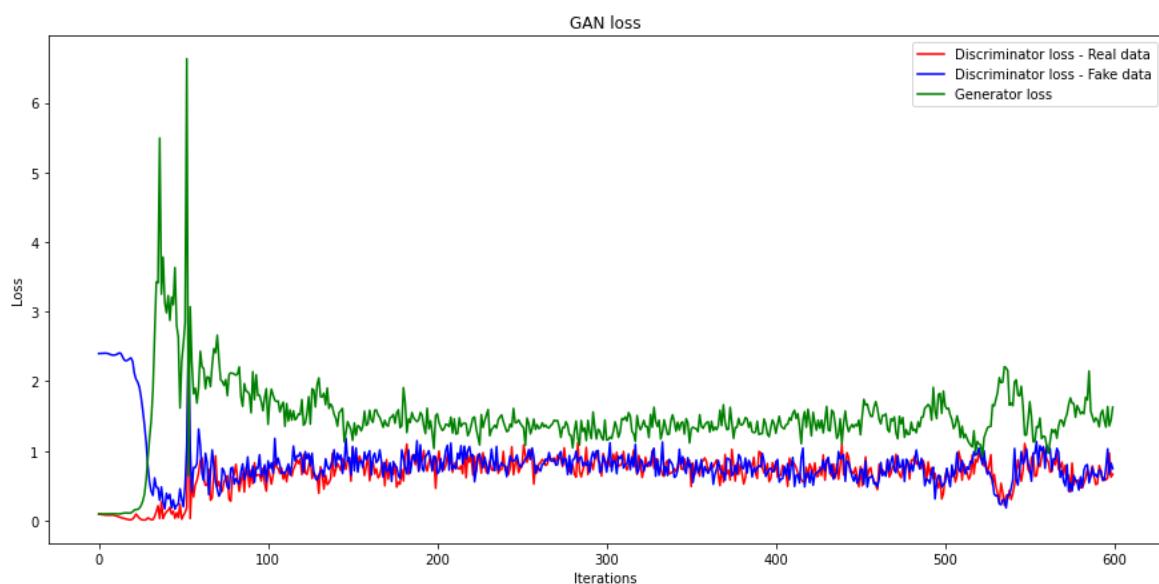
با تعریف بخش‌های مختلف شبکه و نیز تولید داده‌ها، شبکه برای آموزش آماده شده است. در هر مرحله از آموزش، ابتدا یک بچ داده برای آموزش `supervised` بخش `C` انتخاب می‌شود و وزن‌های آن آپدیت می‌شوند. سپس یک بچ داده‌ی واقعی انتخاب و به همان میزان داده‌ی جعلی تولید می‌شود تا بخش `D` بتواند آموزش ببیند. در نهایت نیز به اندازه یک بچ داده (بردار) از فضای `latent` انتخاب می‌شود تا بخش `G` شبکه‌ی `GAN` بتواند آموزش ببیند. عملکرد شبکه در هر چند ایپاک ذخیره می‌شود و تصاویر تولیدی آن تولید می‌شود.

در بخش بعدی، به آموزش شبکه‌ای که توضیح داده شد می‌پردازیم.

ب) آموزش شبکه‌ی Semi-Supervised GAN

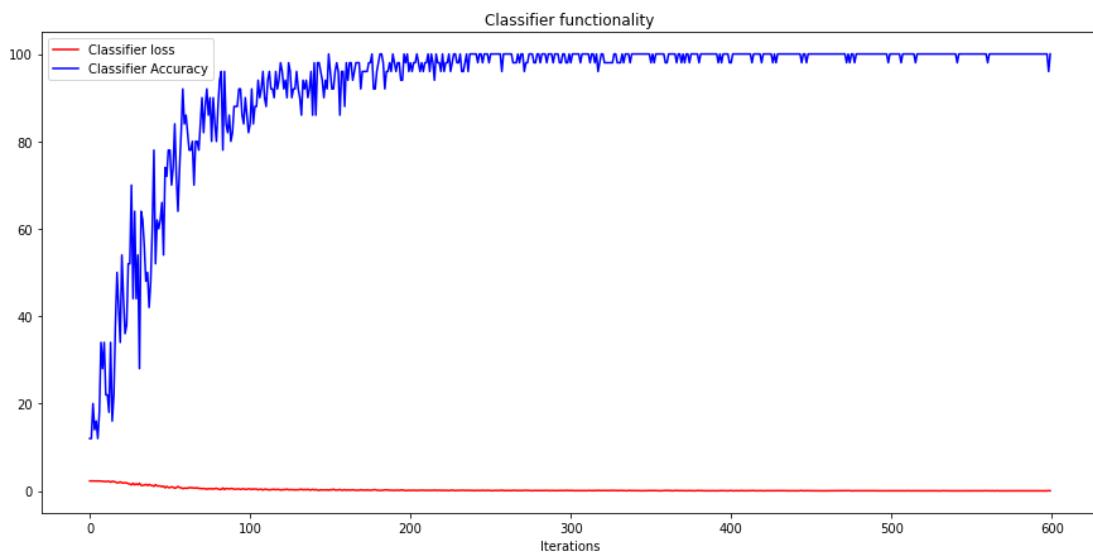
در این بخش، ۱۰۰ داده‌ی برچسبدار در نظر می‌گیریم بدین صورت که از هر کلاس، دقیقاً ۱۰ داده داشته باشیم. تمامی مراحل در بخش الف توضیح داده شده است و تنها کافی است تا کد اجرا شود. در هر ۲۰۰ ایتریشن، یک بار عملکرد مدل ذخیره می‌شود و عکس‌هایی توسط آن تولید می‌شود. همچنین نمودارهای loss برای بخش‌های جنریتور، دیسکریمینیتور بر روی داده‌های واقعی، دیسکریمینیتور بر روی داده‌های جعلی، و در نهایت کلسیفایر ترسیم می‌شود. در نهایت دقت بر روی داده‌های تست نیز ارزیابی می‌شود.

نمودار loss برای اجزای مختلف شبکه بدین شکل است:



شکل ۵: نمودار loss برای اجزای مختلف شبکه – آموزش شبکه‌ی SGAN با ۱۰۰ داده برچسبدار

همچنین عملکرد بخش کلسیفایر به صورت زیر است:



شکل ۶: عملکرد بخش کلیسیفایر – آموزش شبکه‌ی SGAN با ۱۰۰ داده برچسبدار

مشاهده می‌شود که خطای جنریتور در ابتدا کم بوده است، اما پس از طی کردن چند ایتریشن، مقدار خطای آن بالا می‌رود که هدف شبکه هم همین است. از طرفی خطای بخش D در تشخیص داده‌های جعلی کاهش می‌یابد و بهتر عمل می‌کند. در نهایت با تست بر روی داده‌های تست، دقت کلیسیفایر برابر با ۸۳.۷۸٪ بدست آمد.

داده‌های تولیدی پس از طی کردن هر ۲۰۰ ایتریشن به صورت زیر است:

۹	۰	۷	۷	۴	۵	۷	۰	۳	۵	۱
۱	۵	۸	۶	۶	۳	۵	۱	۱	۲	۲
۷	۲	۲	۶	۱	۲	۶	۴	۳	۶	۶
۵	۳	۵	۹	۹	۹	۰	۰	۲	۰	۰
۶	۷	۷	۹	۴	۵	۲	۱	۱	۹	۹
۸	۱	۱	۶	۴	۵	۳	۰	۰	۳	۳
۴	۶	۶	۹	۵	۸	۸	۰	۰	۶	۶
۵	۴	۲	۹	۷	۷	۲	۲	۲	۷	۷
۲	۳	۷	۶	۱	۲	۱	۲	۲	۶	۶
۳	۶	۷	۲	۵	۶	۲	۲	۰	۱	۱

شکل ۷: داده‌های تولید شده توسط شبکه GAN آموزش دیده با ۱۰۰ داده برچسبدار – پس از ۲۰۰ ایتریشن

۷	۵	۲	۳	۳	۷	۸	۹	۴	۶
۵	۹	۳	۵	۴	۶	۱	۲	۰	۸
۴	۱	۱	۵	۶	۷	۲	۴	۰	۰
۳	۲	۴	۷	۸	۹	۵	۶	۵	۷
۲	۹	۵	۶	۴	۳	۰	۹	۸	۶
۱	۹	۰	۳	۶	۷	۳	۰	۷	۹
۰	۰	۵	۹	۷	۲	۵	۹	۹	۷
۶	۹	۳	۷	۵	۰	۷	۹	۵	۳
۵	۶	۰	۳	۵	۴	۲	۵	۶	۷
۴	۷	۳	۳	۰	۸	۱	۳	۴	۵

شکل ۸: داده‌های تولید شده توسط شبکه GAN آموزش دیده با ۱۰۰ داده برچسبدار – پس از ۴۰۰ ایتریشن

۹	۱	۳	۶	۹	۳	۴	۷	۹	۶
۲	۹	۴	۵	۱	۷	۸	۳	۵	۹
۰	۴	۴	۶	۱	۷	۸	۰	۴	۴
۱	۷	۵	۹	۲	۴	۸	۲	۹	۰
۳	۸	۷	۷	۸	۳	۷	۶	۹	۲
۴	۳	۵	۴	۳	۳	۹	۶	۷	۲
۴	۷	۳	۹	۱	۳	۳	۳	۲	۵
۵	۵	۲	۱	۸	۷	۲	۳	۶	۵
۴	۹	۷	۳	۰	۸	۱	۳	۲	۹
۱	۳	۷	۵	۰	۲	۱	۳	۴	۵

شکل ۹: داده‌های تولید شده توسط شبکه GAN آموزش دیده با ۱۰۰ داده برچسبدار – پس از ۶۰۰ ایتریشن

همچنین برای تست بیشتر، شبکه تا ۱۲۰۰ ایتریشن آموزش دید و تصویر تولیدی در آن مرحله به صورت زیر است:

۵	۳	۷	۲	۳	۱	۹	۹	۷	۶
۶	۹	۸	۴	۹	۸	۵	۹	۳	۰
۱	۴	۵	۳	۰	۷	۱	۴	۱	۷
۵	۴	۳	۶	۲	۲	۳	۳	۹	۵
۴	۷	۱	۱	۴	۸	۴	۹	۵	۳
۶	۱	۸	۶	۳	۶	۲	۴	۸	۳
۲	۸	۷	۵	۳	۲	۹	۸	۶	۵
۳	۱	۹	۳	۲	۶	۵	۳	۳	۸
۴	۹	۷	۵	۹	۱	۱	۳	۹	۶
۷	۰	۴	۷	۹	۷	۸	۲	۳	۴

شکل ۱۰: داده‌های تولید شده توسط شبکه GAN آموزش دیده با ۱۰۰ داده برچسبدار – پس از ۱۲۰۰ ایتریشن

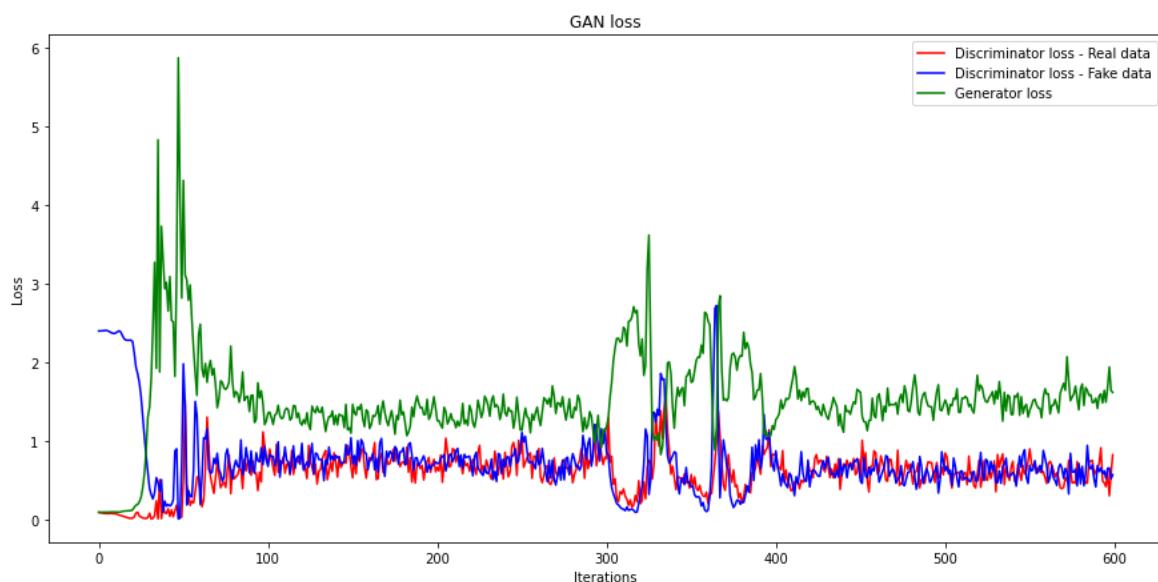
همانطور که مشاهده می‌شود، شبکه‌ی مولد هم همانند بخش‌های C و D با آموزش بیشتر، نمونه‌های بهتری تولید می‌کند. البته که باید مراقب اورفیت شدن شبکه بود که در آن صورت، دقت کلسیفایر بیشتر شده اما کیفیت تصاویر تولیدی کاهش می‌یابد. همچنین واضح است که با افزایش تعداد آموزش‌ها تا حدی معقول، کیفیت تصاویر تولیدی بسیار بهتر از نمونه‌های بالا می‌شود. اما به دلیل محدودیت‌های سخت افزاری، به همین تعداد ایتریشن بستنده شد.

ج) آموزش شبکه Classifier و تشکیل جدول مقایسه

در این بخش برای تشکیل جدول، ابتدا با تعداد ۱۰۰۰، ۵۰ و ۲۵ داده برچسبدار شبکه GAN را آموزش می‌دهیم و سپس بخش D و G آنرا خاموش کرده و تنها C را آموزش می‌دهیم (مانند اینکه تنها داریم یک شبکه CNN را آموزش می‌دهیم). سپس به مقایسه این حالات می‌پردازیم.

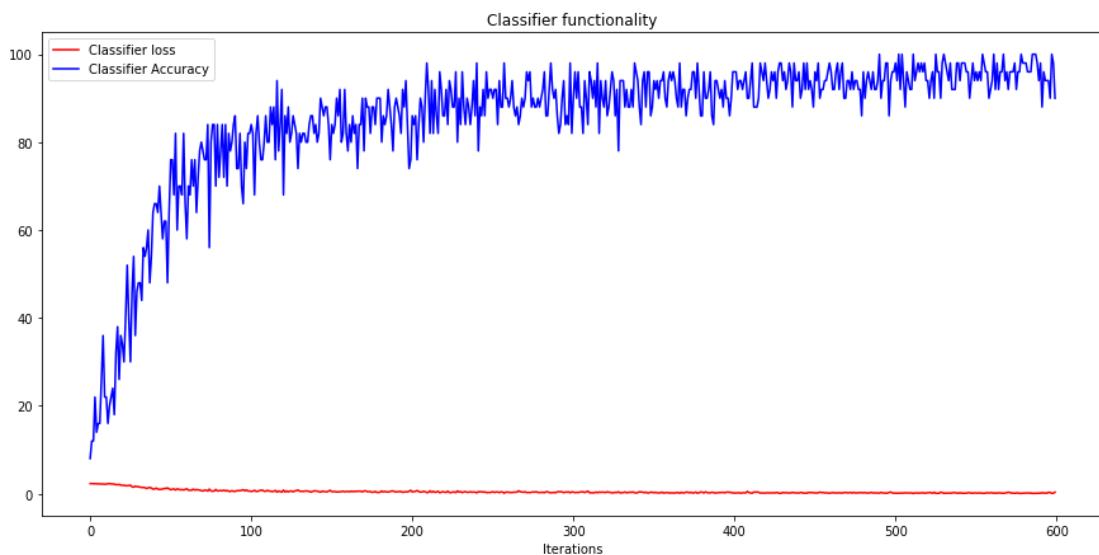
آموزش شبکه GAN با ۱۰۰۰ داده

در این بخش ۱۰۰۰ داده برچسبدار در نظر می‌گیریم و شبکه GAN را همانند بخش قبلی آموزش می‌دهیم. نمودار loss برای اجزای مختلف شبکه بدین شکل است:



شکل ۱۱: نمودار loss برای اجزای مختلف شبکه آموزش شبکه $SGAN$ با ۱۰۰۰ داده برچسبدار

همچنین عملکرد بخش کلسیفایر به صورت زیر است:



شکل ۱۲: عملکرد بخش کلسیفایر – آموزش شبکه‌ی SGAN با ۱۰۰۰ داده برچسبدار

مشاهده می‌شود که خطای جنریتور در ابتدا کم بوده است، اما پس از طی کردن چند ایتریشن، مقدار خطای آن بالا می‌رود که هدف شبکه هم همین است. از طرفی خطای بخش D در تشخیص داده‌های جعلی کاهش می‌یابد و بهتر عمل می‌کند. نکته حائز اهمیت این است که به دلیل تعداد بیشتر داده‌های لیبل دار، شبکه دیرتر از حالت قبل اورفیت شده است. در نهایت با تست بر روی داده‌های تست، دقت کلسیفایر برابر با ۹۴.۰۱٪ بدست آمد. داده‌های تولیدی پس از طی کردن هر ۲۰۰ ایتریشن به صورت زیر است:

۶	۵	۳	۱۱	۶	۸	۵	۳	۷	۲
۴	۲	۱۱	۵	۵	۲	۴	۱	۱	۳
۷	۶	۳	۹	۵	۲	۴	۳	۷	۷
۹	۱	۷	۲	۴	۹	۳	۰	۶	۵
۶	۳	۱	۵	۳	۶	۷	۴	۵	۲
۳	۷	۱	۴	۷	۷	۳	۳	۶	۶
۱	۷	۰	۲	۷	۴	۶	۴	۵	۷
۴	۸	۵	۱	۳	۴	۳	۷	۶	۴
۲	۷	۳	۵	۱	۴	۶	۵	۱	۲
۵	۵	۸	۱	۷	۷	۲	۸	۹	۶

شکل ۱۳: داده‌های تولید شده توسط شبکه GAN آموزش دیده با ۱۰۰۰ داده برچسبدار – پس از ۲۰۰ ایتریشن

5	7	7	1	4	5	9	6	6	6
5	3	5	4	5	6	9	6	1	5
2	7	4	4	8	6	9	2	2	3
6	5	3	7	5	3	6	4	7	3
2	7	5	6	9	7	4	9	9	4
7	5	8	2	5	9	2	4	0	5
8	4	6	5	5	5	6	5	1	0
6	6	5	6	2	9	2	2	3	0
2	3	3	3	6	3	7	4	3	9
9	2	9	5	9	7	6	2	2	5

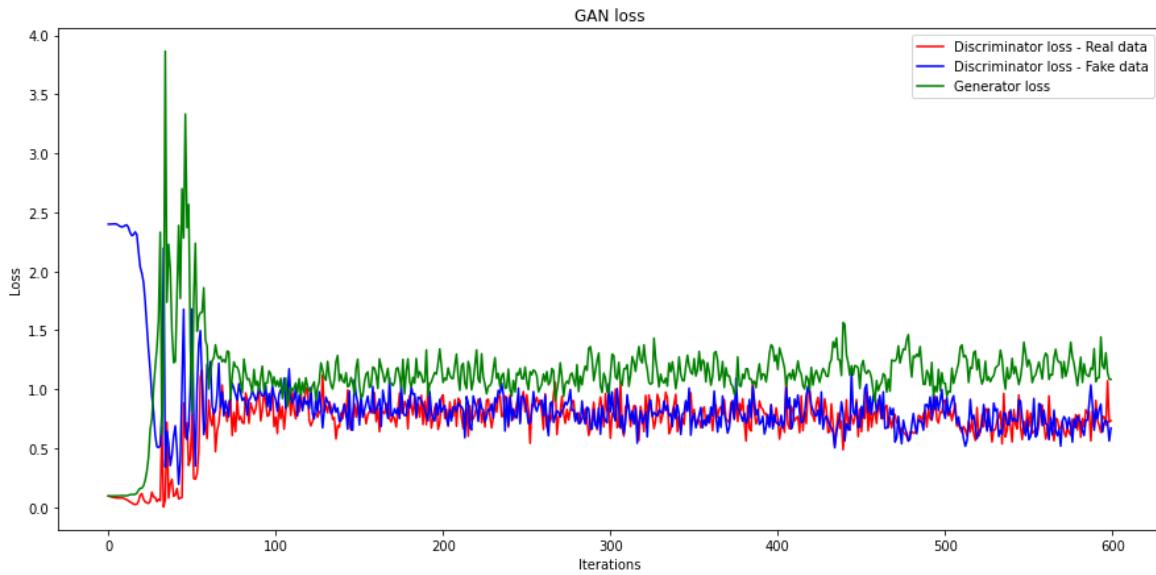
شکل ۱۴: داده‌های تولید شده توسط شبکه GAN آموزش دیده با ۱۰۰۰ داده برچسب‌دار – پس از ۴۰۰ ایتریشن

2	4	2	6	0	9	7	8	5	5
7	4	8	8	8	3	3	?	3	2
7	9	6	9	3	4	2	3	3	8
7	7	9	4	9	0	9	?	4	2
2	4	3	!	3	3	8	2	4	7
5	5	9	7	1	7	8	3	7	5
7	4	2	3	8	8	2	9	5	4
5	2	7	8	9	7	6	7	0	3
7	8	4	3	7	6	2	4	2	7
7	0	9	4	8	0	9	7	3	9

شکل ۱۵: داده‌های تولید شده توسط شبکه GAN آموزش دیده با ۱۰۰۰ داده برچسب‌دار – پس از ۶۰۰ ایتریشن

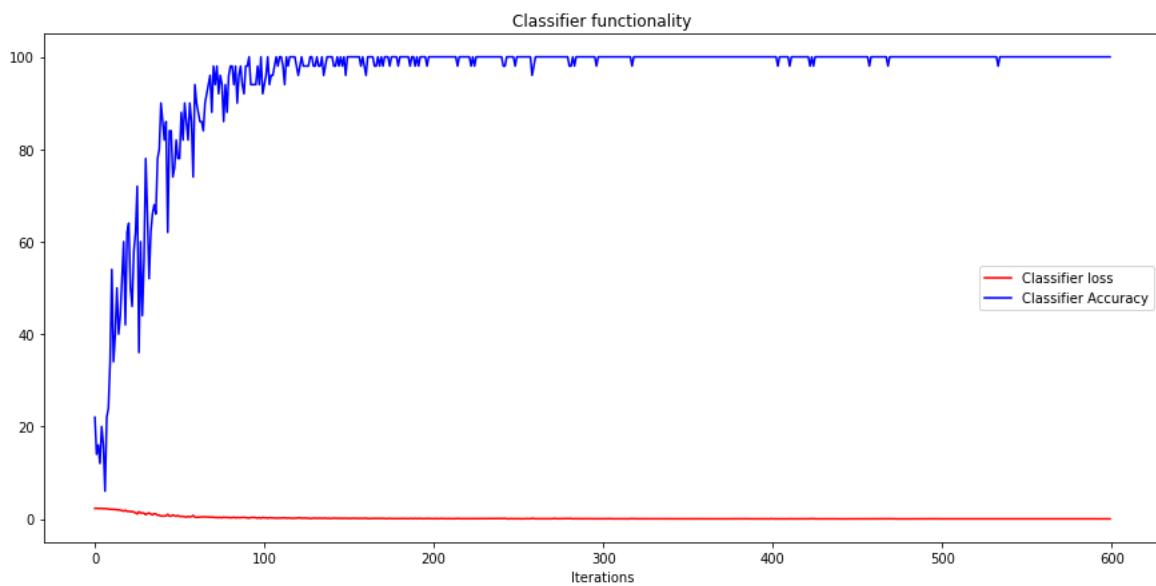
آموزش شبکه GAN با ۵۰ داده

در این بخش تنها ۵۰ داده برچسب‌دار در نظر می‌گیریم و شبکه GAN را همانند بخش قبلی آموزش می‌دهیم. نمودار loss برای اجزای مختلف شبکه بدین شکل است:



شکل ۱۶: نمودار loss برای اجزای مختلف شبکه آموزش شبکه‌ی SGAN با ۵۰ داده برچسب‌دار

همچنین عملکرد بخش کلسیفایر به صورت زیر است:



شکل ۱۷: عملکرد بخش کلسیفایر – آموزش شبکه‌ی SGAN با ۵۰ داده برچسب‌دار

مشاهده می‌شود که خطای جنریتور در ابتدا کم بوده است، اما پس از طی کردن چند ایتریشن، مقدار خطای آن بالا می‌رود که هدف شبکه هم همین است. از طرفی خطای بخش D در تشخیص داده‌های جعلی کاهش می‌یابد و بهتر عمل می‌کند. نکته حائز اهمیت این است که به دلیل تعداد کمتر داده‌های لیبل دار، شبکه زودتر از حالت قبل اورفیت شده است. در نهایت با تست بر روی داده‌های تست، دقت کلسیفایر برابر با ۷۷.۲۲٪ بدست آمد. داده‌های تولیدی پس از طی کردن هر ۲۰۰ ایتریشن به صورت زیر است:

۶	۶	۷	۵	۳	۹	۸	۶	۲	۴
۵	۵	۴	۰	۲	۳	۲	۱	۴	۳
۵	۵	۷	۳	۷	۹	۲	۰	۲	۷
۶	۹	۳	۰	۶	۷	۴	۵	۷	۲
۴	۷	۹	۲	۳	۰	۷	۶	۱	۵
۳	۷	۴	۲	۰	۱	۲	۶	۹	۷
۹	۲	۸	۰	۱	۵	۶	۷	۳	۰
۲	۹	۶	۳	۰	۲	۵	ۢ	۷	۴
۴	۷	۵	۹	۸	۱	۲	۳	۰	۳
۶	۴	۳	۷	۶	۵	۲	۹	۰	۰

شکل ۱۸: داده‌های تولید شده توسط شبکه GAN آموزش دیده با ۵۰ داده برچسبدار – پس از ۲۰۰ ایتریشن

۸	۸	۹	۸	۵	۹	۶	۰	۷	۳
۲	۵	۴	۸	۷	۸	۵	۲	۴	۰
۵	۰	۳	۷	۹	۳	۳	۸	۵	۱
۴	۳	۹	۳	۹	۵	۴	۵	۳	۳
۹	۶	۸	۷	۲	۹	۶	۳	۴	۸
۸	۶	۵	۹	۵	۶	۳	۹	۳	۳
۴	۲	۹	۸	۴	۶	۵	۳	۷	۰
۵	۲	۵	۳	۱	۴	۵	۰	۳	۰
۷	۴	۸	۶	۵	۹	۳	۷	۱	۱
۶	۶	۳	۵	۴	۴	۳	۸	۳	۵

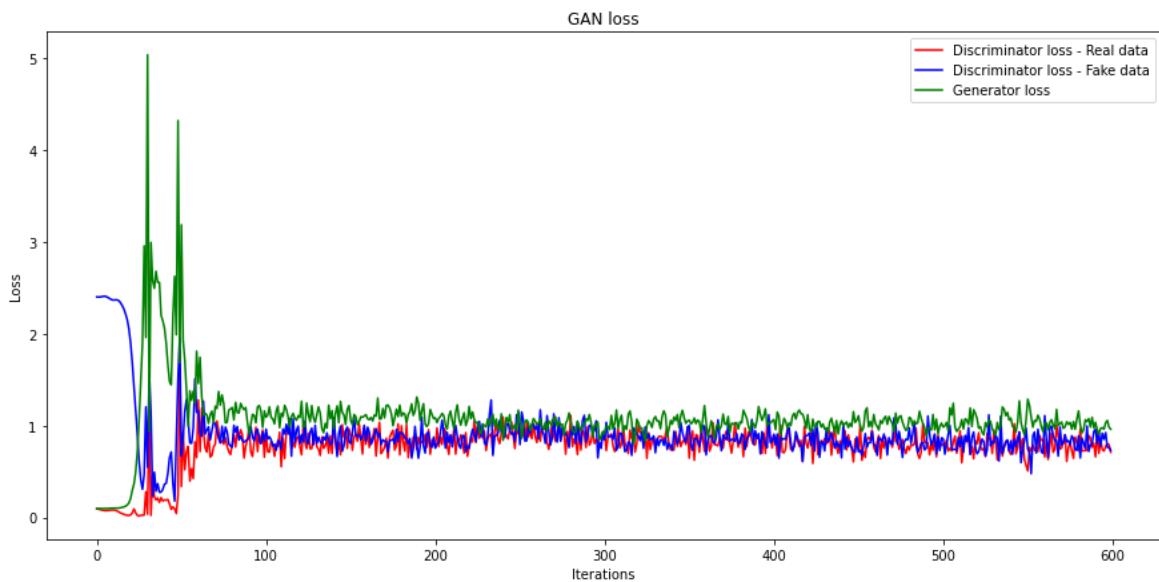
شکل ۱۹: داده‌های تولید شده توسط شبکه GAN آموزش دیده با ۵۰ داده برچسبدار – پس از ۴۰۰ ایتریشن

۸	۸	۶	۹	۷	۳	۸	۹	۷	۲
۸	۹	۷	۴	۶	۲	۹	۲	۳	۱
۹	۳	۲	۶	۱	۵	۴	۷	۴	۹
۳	۲	۳	۵	۴	۷	۵	۶	۴	۳
۹	۹	۵	۶	۵	۴	۹	۲	۹	۵
۶	۷	۴	۵	۴	۲	۴	۴	۹	۷
۷	۳	۴	۹	۰	۷	۴	۳	۹	۵
۳	۸	۰	۹	۳	۹	۳	۷	۵	۳
۱	۸	۹	۰	۶	۷	۹	۱	۸	۸
۶	۳	۴	۹	۱	۲	۹	۹	۰	۰

شکل ۲۰: داده‌های تولید شده توسط شبکه GAN آموزش دیده با ۵۰ داده برچسبدار – پس از ۶۰۰ ایتریشن

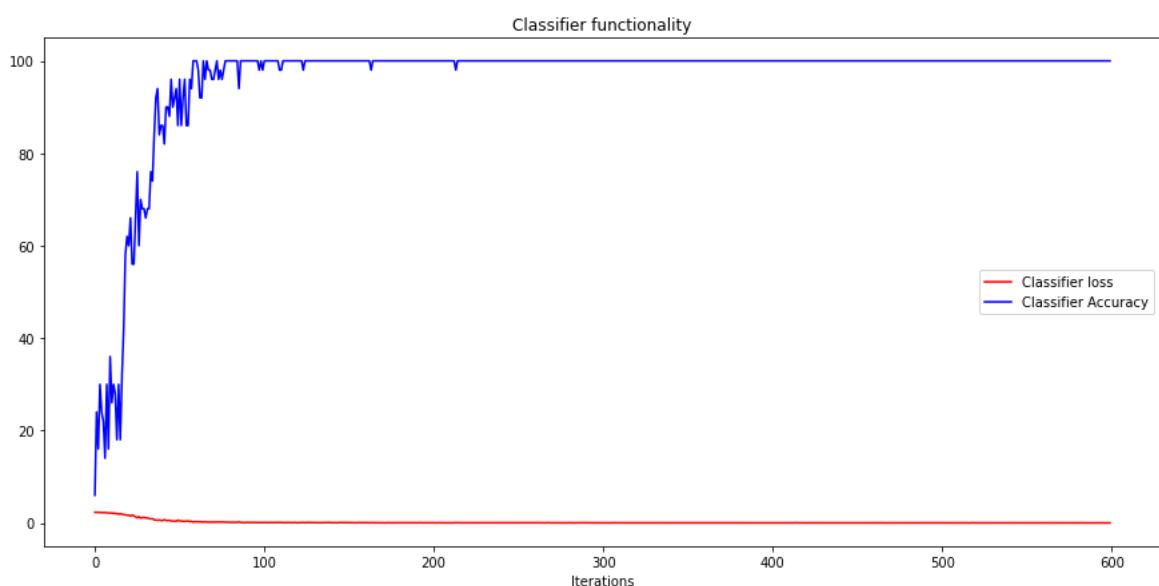
آموزش شبکه *GAN* با ۲۵ داده

در این بخش تنها ۲۵ داده برچسبدار در نظر می‌گیریم و شبکه *GAN* را همانند بخش قبلی آموزش می‌دهیم. نمودار loss برای اجزای مختلف شبکه بدین شکل است:



شکل ۲۱: نمودار loss برای اجزای مختلف شبکه – آموزش شبکه‌ی *SGAN* با ۲۵ داده برچسبدار

همچنین عملکرد بخش کلسیفایر به صورت زیر است:



شکل ۲۲: عملکرد بخش کلسیفایر – آموزش شبکه‌ی *SGAN* با ۲۵ داده برچسبدار

مشاهده می‌شود که خطای جنریتور در ابتدا کم بوده است، اما پس از طی کردن چند ایتریشن، مقدار خطای آن بالا می‌رود که هدف شبکه هم همین است. از طرفی خطای بخش D در تشخیص داده‌های جعلی کاهش می‌یابد و بهتر عمل می‌کند. نکته حائز اهمیت این است که به دلیل تعداد کمتر داده‌های

لیبل دار، شبکه زودتر از حالت قبل اورفیت شده است. در نهایت با تست بر روی داده های تست، دقت کلسیفایر برابر با ۶۵.۷۸٪ بدست آمد. داده های تولیدی پس از طی کردن هر ۲۰۰ ایتریشن به صورت زیر است:

۹	۵	۲	۶	۳	۷	۵	۴	۶	۲
۷	۵	۷	۸	۶	۲	۷	۵	۶	۹
۳	۶	۹	۶	۳	۹	۳	۷	۵	۹
۸	۲	۷	۸	۵	۴	۷	۶	۹	۷
۱	۲	۳	۱	۶	۶	۷	۲	۵	۴
۴	۳	۵	۲	۷	۱	۴	۵	۸	۱
۶	۳	۳	۵	۳	۳	۴	۶	۶	۳
۵	۵	۶	۷	۲	۳	۸	۳	۲	۵
۹	۸	۶	۴	۷	۹	۳	۵	۱	۸
۰	۹	۴	۹	۴	۳	۳	۰	۶	۵

شکل ۲۳: داده های تولید شده توسط شبکه SGAN آموزش دیده با ۲۵ داده برچسب دار – پس از ۲۰۰ ایتریشن

۱	۶	۴	۳	۴	۲	۳	۷	۳	۰
۸	۴	۳	۳	۷	۳	۵	۷	۵	۹
۷	۵	۳	۴	۴	۶	۶	۹	۶	۶
۹	۷	۴	۲	۳	۳	۳	۸	۰	۵
۰	۳	۶	۸	۴	۷	۶	۴	۲	۷
۵	۵	۸	۳	۳	۵	۵	۷	۳	۷
۶	۷	۲	۴	۷	۹	۴	۶	۰	۶
۶	۹	۴	۵	۲	۶	۶	۵	۵	۵
۳	۷	۰	۰	۵	۳	۷	۳	۴	۹
۰	۲	۵	۲	۴	۹	۶	۰	۷	۰

شکل ۲۴: داده های تولید شده توسط شبکه SGAN آموزش دیده با ۲۵ داده برچسب دار – پس از ۴۰۰ ایتریشن

۱	۸	۴	۳	۳	۲	۵	۱	۷	۶
۳	۹	۵	۴	۸	۲	۶	۸	۴	۴
۷	۱	۱	۶	۵	۵	۶	۶	۴	۴
۷	۹	۶	۷	۶	۷	۶	۳	۷	۷
۴	۶	۴	۳	۰	۷	۷	۸	۲	۷
۵	۵	۷	۸	۵	۷	۲	۷	۲	۱
۰	۱	۷	۹	۳	۶	۶	۴	۵	۰
۶	۷	۵	۶	۵	۱	۷	۵	۵	۶
۳	۹	۳	۴	۷	۵	۲	۳	۶	۶
۰	۷	۵	۴	۵	۷	۳	۶	۷	۷

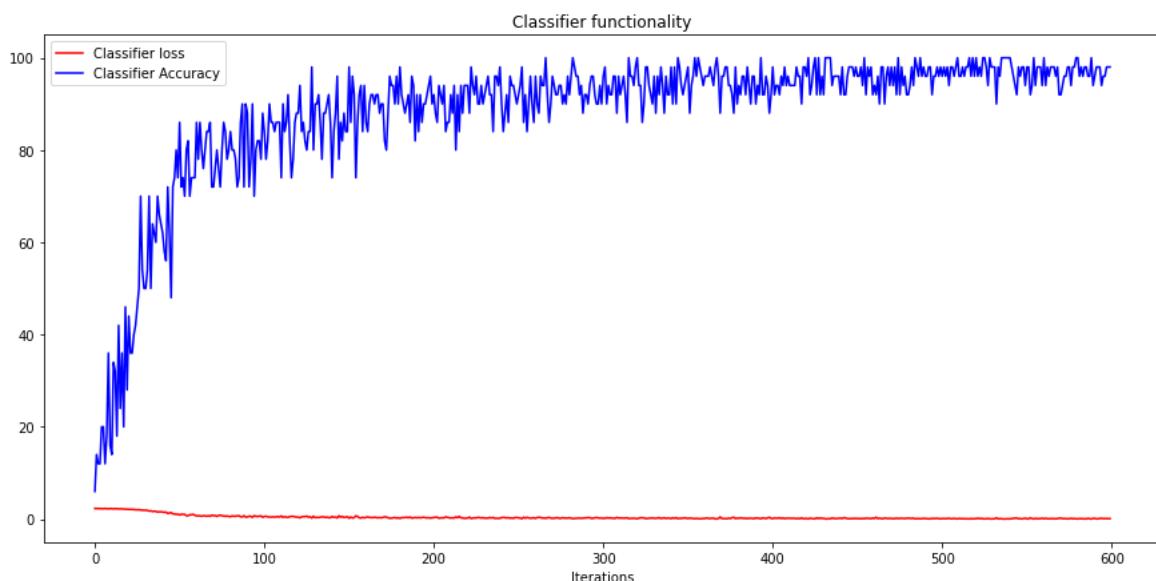
شکل ۲۵: داده های تولید شده توسط شبکه SGAN آموزش دیده با ۲۵ داده برچسب دار – پس از ۶۰۰ ایتریشن

با دقت به تصاویر، مشاهده می شود که کیفیت آنها نیز نسبت به حالت ۱۰۰ تصویر لیبل دار، افت داشته است.

حال پس از آموزش شبکه‌های GAN در سه حالت، این بار تنها بخش *Classifier* آن‌ها را آموزش می‌دهیم و نتایج را مقایسه می‌کنیم.

آموزش شبکه CNN با ۱۰۰۰ داده

در این بخش ۱۰۰۰ داده برچسبدار در نظر می‌گیریم و شبکه CNN را آموزش می‌دهیم. بر خلاف قسمت قبلی، دیگر بخش G و D را کاری نداشته و آن‌ها را از فرایند آموزش کنار می‌گذاریم. بنابراین دیگر خطای G و D را محاسبه نکرده و تصاویر تولیدی نیز مفهومی ندارند. عملکرد بخش کلسیفایر به صورت زیر است:

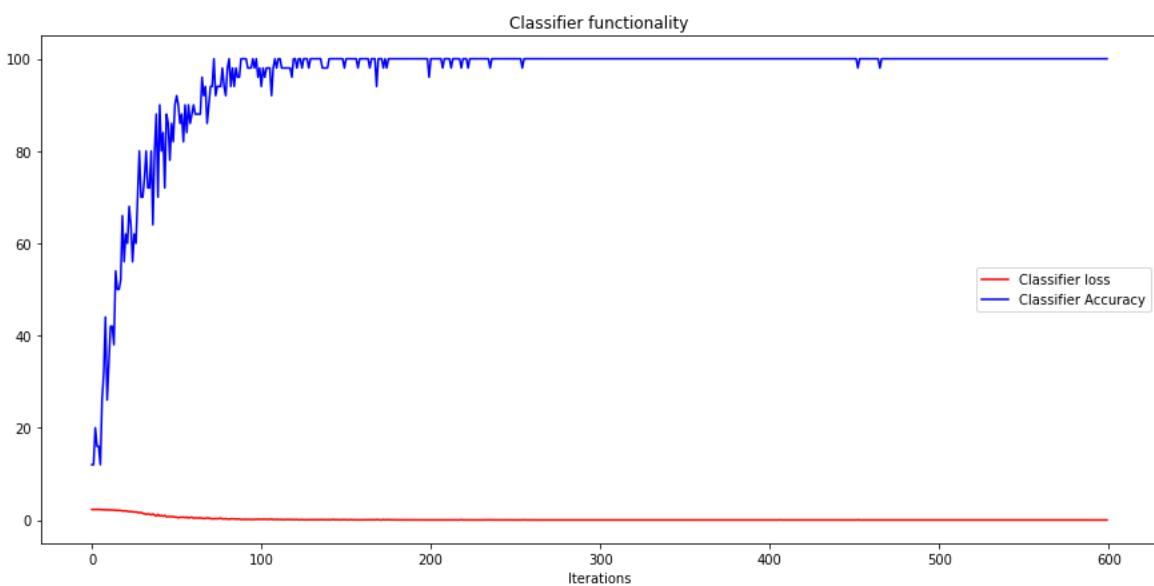


شکل ۲۶: عملکرد بخش کلسیفایر – آموزش شبکه‌ی CNN با ۱۰۰۰ داده برچسبدار

مشاهده می‌شود که تقریباً پس از ۴۰۰ ایتریشن، شبکه بر روی داده‌های آموزشی اورفیت می‌شود. اما برای مقایسه با حالت قبلی، تا ۶۰۰ ایتریشن پیش رفته‌ایم. در نهایت با تست بر روی داده‌های تست، دقت کلسیفایر برابر با ۹۱.۹۲٪ بدست آمد. ذکر این نکته نیز لازم است که به دلیل آنکه فقط بخش CNN آموزش می‌بیند، زمان مورد نیاز برای آموزش شبکه بسیار کمتر از حالت GAN می‌شود.

آموزش شبکه CNN با ۱۰۰ داده

در این بخش تنها ۱۰۰ داده برچسبدار در نظر می‌گیریم و شبکه CNN را آموزش می‌دهیم. عملکرد بخش کلسیفایر به صورت زیر است:

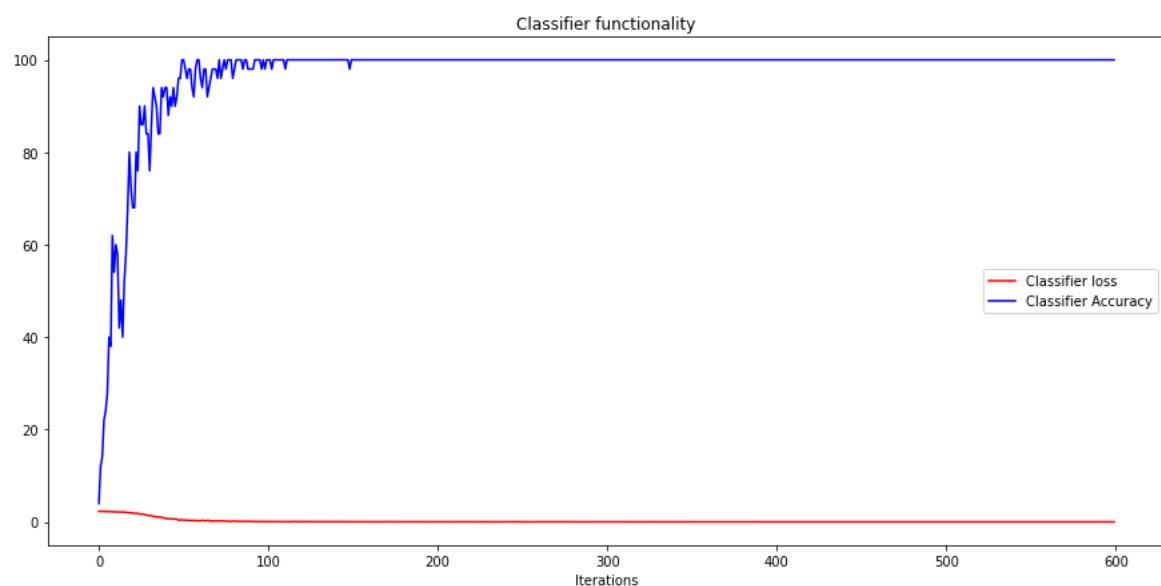


شکل ۲۷: عملکرد بخش کلسیفایر – آموزش شبکه‌ی CNN با ۱۰۰ داده برچسب‌دار

مشاهده می‌شود که تقریباً پس از ۱۰۰ ایتریشن، شبکه بر روی داده‌های آموزشی اورفیت می‌شود. اما برای مقایسه با حالت قبلی، تا ۶۰۰ ایتریشن پیش رفته‌ایم. در نهایت با تست بر روی داده‌های تست، دقت کلسیفایر برابر با ۷۸.۷۴٪ بدست آمد.

آموزش شبکه CNN با ۵۰ داده

در این بخش تنها ۵۰ داده برچسب‌دار در نظر می‌گیریم و شبکه CNN را آموزش می‌دهیم. عملکرد بخش کلسیفایر به صورت زیر است:

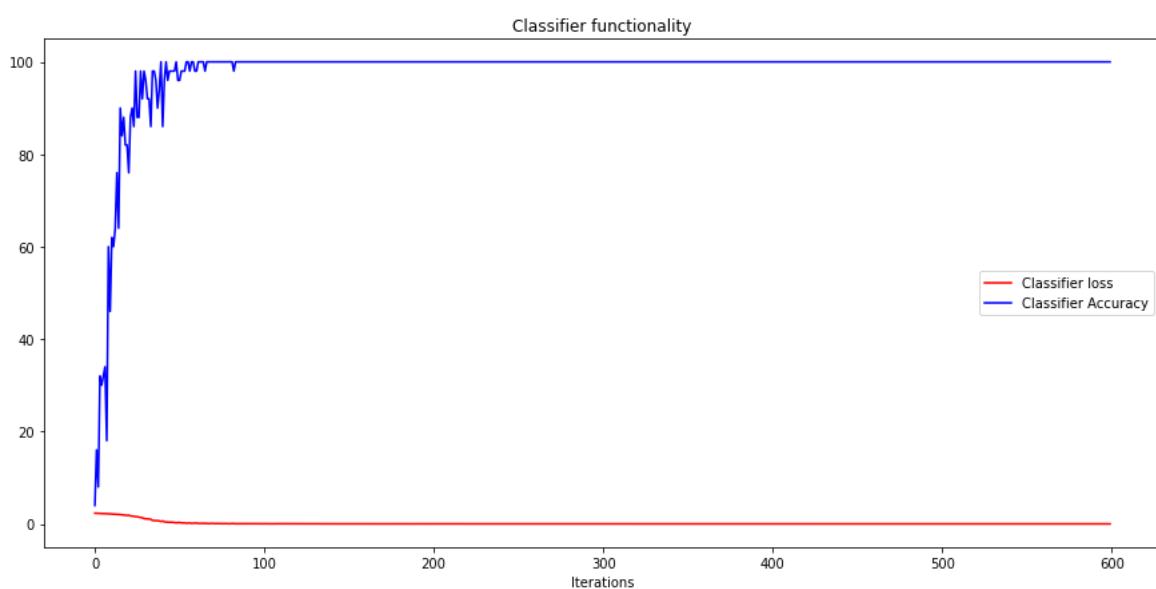


شکل ۲۸: عملکرد بخش کلسیفایر – آموزش شبکه‌ی CNN با ۵۰ داده برچسب‌دار

مشاهده می‌شود که تقریباً پس از ۸۰ ایتریشن، شبکه بر روی داده‌های آموزشی اورفیت می‌شود. اما برای مقایسه با حالت قبلی، تا ۶۰۰ ایتریشن پیش رفته‌ایم. در نهایت با تست بر روی داده‌های تست، دقت کلسیفایر برابر با ۷۰.۹۸٪ بدست آمد.

آموزش شبکه CNN با ۲۵ داده

در این بخش تنها ۲۵ داده برچسبدار در نظر می‌گیریم و شبکه CNN را آموزش می‌دهیم. عملکرد بخش کلسیفایر به صورت زیر است:



شکل ۲۹: عملکرد بخش کلسیفایر – آموزش شبکه‌ی CNN با ۲۵ داده برچسبدار

مشاهده می‌شود که تقریباً پس از ۶۰ ایتریشن، شبکه بر روی داده‌های آموزشی اورفیت می‌شود. اما برای مقایسه با حالت قبلی، تا ۶۰۰ ایتریشن پیش رفته‌ایم. در نهایت با تست بر روی داده‌های تست، دقت کلسیفایر برابر با ۶۱.۶۱٪ بدست آمد.

جدول مقایسه

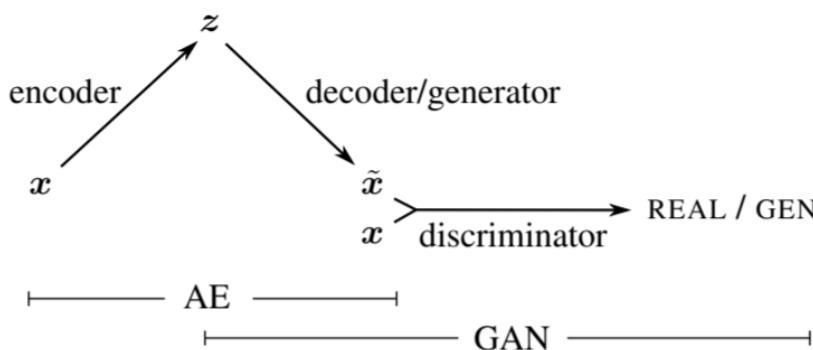
در این بخش، می‌خواهیم گزاره‌ای را که در ابتدا آورده شد (مبنی بر اینکه بهبود D می‌تواند موجب بهبود C شده و بالعکس) را راستی‌آزمایی کنیم. در بخش‌های قبل برای دو حالت شبکه CNN و GAN ، با تعداد ۲۵، ۵۰، ۱۰۰ و ۱۰۰۰ داده آموزشی، سعی کردیم شبکه‌ها را آموزش دهیم. نتایج شبکه بر روی داده‌های تست به صورت خلاصه در جدول زیر آورده شده است:

$SGAN$ accuracy (%)	CNN accuracy (%)	تعداد نمونه‌های برچسبدار
۹۴.۰۱	۹۱.۹۲	۱۰۰۰
۸۳.۷۸	۷۸.۷۴	۱۰۰
۷۷.۲۲	۷۰.۹۸	۵۰
۶۵.۷۸	۶۱.۶۱	۲۵

همان‌طور که از جدول مشخص است در تعداد سمپل‌های لیبل‌دار مشابه، عملکرد شبکه $SGAN$ بعضاً تا ۷ درصد بهتر بوده است. همچنین در تعداد سمپل‌های پایین‌تر که شبکه به سرعت اورفیت می‌شود، برتری $SGAN$ بیشتر مشهود است نسبت به موقعی که تعداد سمپل‌های لیبل‌دار زیادی در اختیار داریم (تقرباً ۲۰٪ درصد اختلاف در حالت ۱۰۰۰ سمپل). بنابراین می‌توانیم بگوییم گزاره مورد آزمون، درست بوده است.

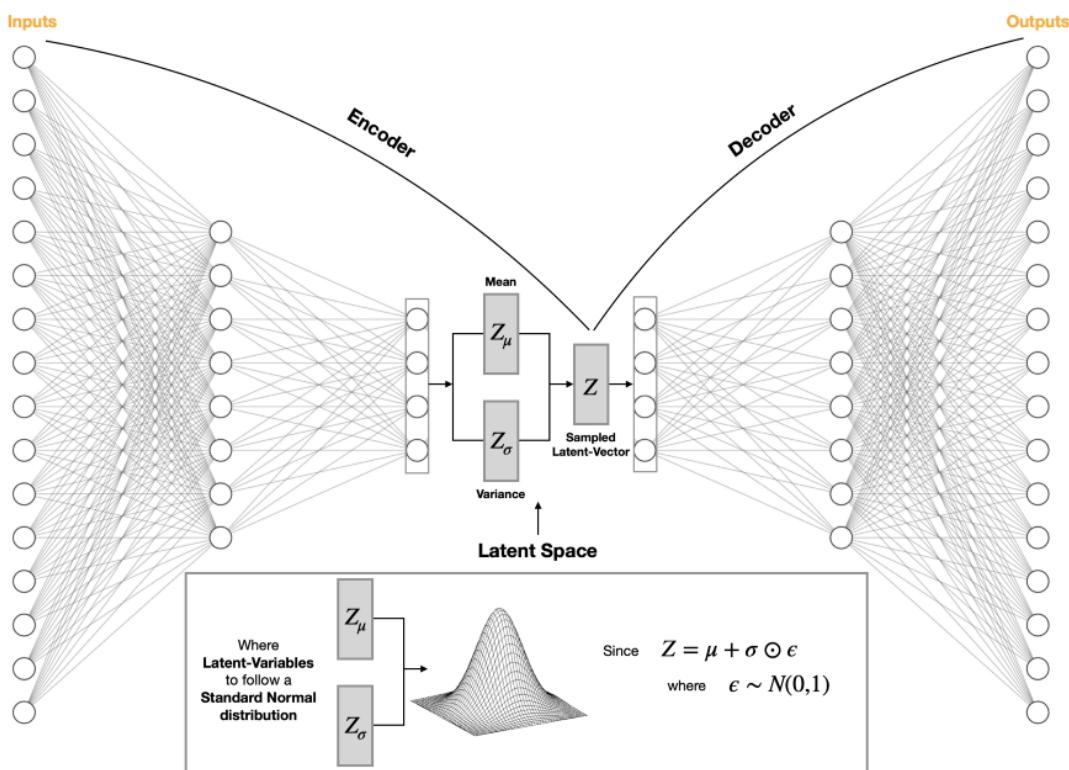
د) استفاده از Variational autoencoder به عنوان Generator

در این بخش از یک VAE به عنوان جنریتور استفاده می‌شود. این ایده از مقاله‌ای به نام "Autoencoding beyond pixels using a learned similarity metric" برآمده‌است و در شکل زیر می‌توان ایده کلی آنرا مشاهده کرد:



شکل ۳۰: عملکرد کلی شبکه‌ی **VAEGAN**

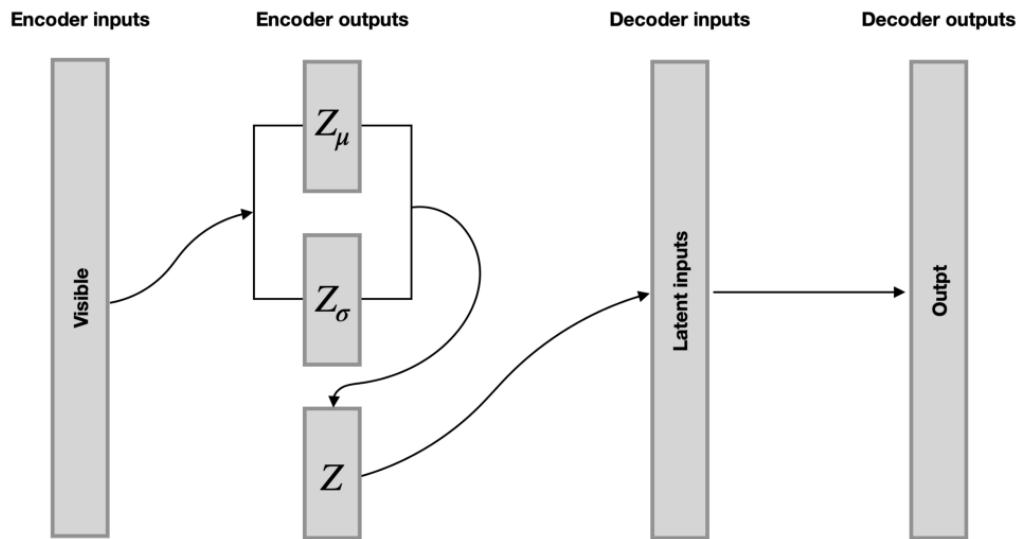
ابتدا ساختار یک *VAE* بررسی می‌شود و سپس به اجرای کد می‌پردازیم. به طور کلی، می‌توان گفت یک *VAE* ساختاری تقریباً مشابه با یک اتوانکودر دارد. معماری این شبکه در تصویر زیر ترسیم شده است:



شکل ۳۱: ساختار شماتیک *VAE*

تفاوت اصلی اتوانکودر با *VAE* در این نکته است که فضای *latent* شبکه *VAE*، پیوسته بوده و به جای *node* های گستته، ورودی ها به یک توزیع نرمال مپ می‌شوند که میانگین و واریانس آن را شبکه در طی آموزش یاد می‌گیرد. سپس، بردار Z از این فضای پیوسته با میانگین $Z\mu$ و واریانس $Z\sigma$ سمپل شده و به بخش دیکودر می‌رود تا خروجی ها را پیش‌بینی کند. پیوسته بودن *VAE* این امکان را به ما می‌دهد که از بخش های مختلف آن برای تولید نمونه های جدید داده برداری کنیم. در تصویر زیر، می‌توان دید در فضای گستته امکان اینکه مدل آموزش ببیند تا تفاوت یا شباهات میان نقاط را بفهمد و داده های معنادار تولید کند، وجود ندارد. اما در *VAE* ها، توزیع داده ها به صورت پیوسته امکان انتقال در این فضا به نقاط مختلف را می‌دهد و با سمپل کردن از نقاط نزدیک به یک توزیع، می‌توان داده هایی نزدیک به آن تولید کرد.

یک شبکه *VAE* دارای دو بخش انکودر و دیکودر است. بخش انکودر دارای یک خروجی میانگین و یک خروجی واریانس است، سپس به این خروجی یکتابع سمپلینگ اعمال می‌شود که با کمک متغیر Z اپسیلون (عددی برآمده از توزیع نرمال با میانگین ۰ و واریانس ۱)، یک بردار گستته *latent* به اسم *latent* تولید می‌کند. سپس این بردار توسط لایه هایی میانی آپ سمپل شده و خروجی که مانند ابعاد عمس اصلی است (۲۸*۲۸) تولید می‌کند. فرایند توضیح داده شده در تصویر زیر مشخص است:



شکل ۳۲: مسیر جریان اطلاعات از ورودی تا خروجی **VAE**

در ساختار انکودر و دیکودر، می‌توان از لایه‌های مختلفی استفاده کرد. لایه‌های کانولوشن و دی‌کانولوشن و یا لایه‌های *dense* عادی. به دلیل اینکه ابعاد تصویر خیلی بزرگ نیست، هر دوی این حالات را می‌توان استفاده کرد. در این تمرین، از لایه‌های خطی استفاده شده است. معماری بخش انکودر به شکل زیر است:

Layer (type)	Output Shape	Param #	Connected to
=====			
Encoder-Input-Layer (InputLayer)	[None, 784]	0	[]
Encoder-Hidden-Layer-1 (Dense)	(None, 64)	50240	['Encoder-Input-Layer[0][0]']
Encoder-Hidden-Layer-2 (Dense)	(None, 16)	1040	['Encoder-Hidden-Layer-1[0][0]']
Encoder-Hidden-Layer-3 (Dense)	(None, 8)	136	['Encoder-Hidden-Layer-2[0][0]']
Z-Mean (Dense)	(None, 2)	18	['Encoder-Hidden-Layer-3[0][0]']
Z-Log-Sigma (Dense)	(None, 2)	18	['Encoder-Hidden-Layer-3[0][0]']
Z-Sampling-Layer (Lambda)	(None, 2)	0	['Z-Mean[0][0]', 'Z-Log-Sigma[0][0]']
=====			
Total params: 51,452			
Trainable params: 51,452			
Non-trainable params: 0			

شکل ۳۳: معماری بخش انکودر **VAE**

همچنین معماری بخش دیکودر به شکل زیر است:

Layer (type)	Output Shape	Param #
Input-Z-Sampling (InputLayer)	[None, 2]	0
Decoder-Hidden-Layer-1 (Dense)	(None, 8)	24
Decoder-Hidden-Layer-2 (Dense)	(None, 16)	144
Decoder-Hidden-Layer-3 (Dense)	(None, 64)	1088
Decoder-Output-Layer (Dense)	(None, 784)	50960
reshape (Reshape)	(None, 28, 28, 1)	0
Total params:	52,216	
Trainable params:	52,216	
Non-trainable params:	0	

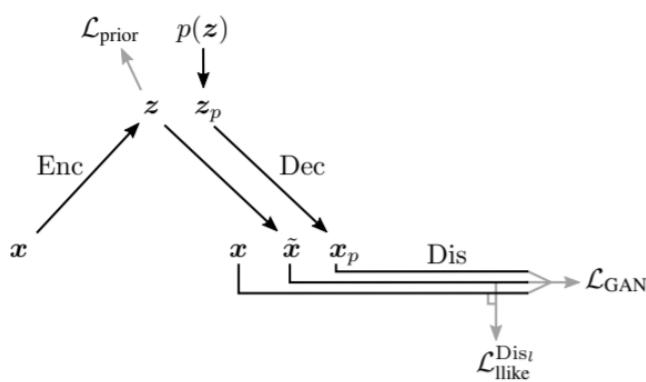
شکل ۳۴: معماری بخش دیکودر VAE

معماری بخش D/C نیز همانند بخش‌های قبلی این سوال است.

برای آموزش شبکه، نیازمند آن هستیم که یک تابع خطای مناسب تعریف کنیم. برای بخش VAE سه مولفه‌ی خطای GAN و خطای بازسازی را در نظر می‌گیریم. خطای بازسازی را در نظر می‌گیریم. خطای GAN نیز بدین شکل تعریف می‌شود:

$$\begin{aligned} \mathcal{L}_{GAN} = & \log(\text{Dis}(x)) + \log(1 - \text{Dis}(\text{Dec}(z))) \\ & + \log(1 - \text{Dis}(\text{Dec}(\text{Enc}(x)))) \end{aligned}$$

فرایند کلی محاسبه‌ی خطای زیر آمده است:



شکل ۳۵: محاسبه‌ی خطای شبکه‌ی VAE GAN

همان‌طور که واضح است، خطای KL (\mathcal{L}_{prior}) با استفاده از توزیع z محاسبه می‌شود:

$$\mathcal{L}_{prior} = D_{KL}(q(z|x) \| p(z))$$

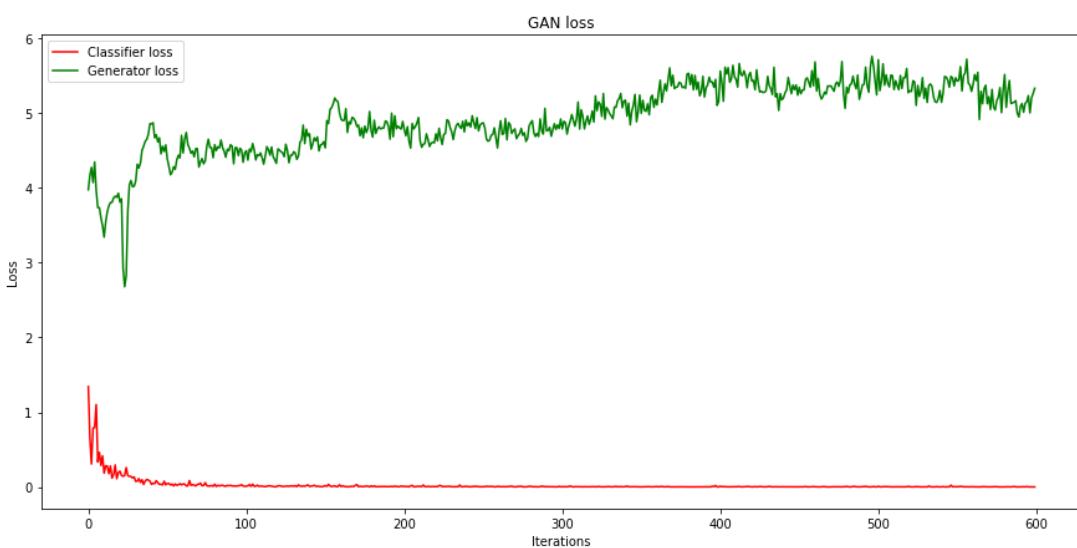
خطای GAN از ترکیب ورودی شبکه و دیکود شده Z و Zp ساخته می‌شود و خطای $\mathcal{L}_{\text{llike}}^{\text{pixel}}$ نیز که با استفاده از دیکود شده Z و ورودی x محاسبه می‌شود، به شکل زیر تعریف شده است:

$$\mathcal{L}_{\text{llike}}^{\text{pixel}} = -\mathbb{E}_{q(\mathbf{z}|\mathbf{x})}[\log p(\mathbf{x}|\mathbf{z})]$$

پس از این تعریف‌ها، به سراغ آموزش شبکه می‌رویم.

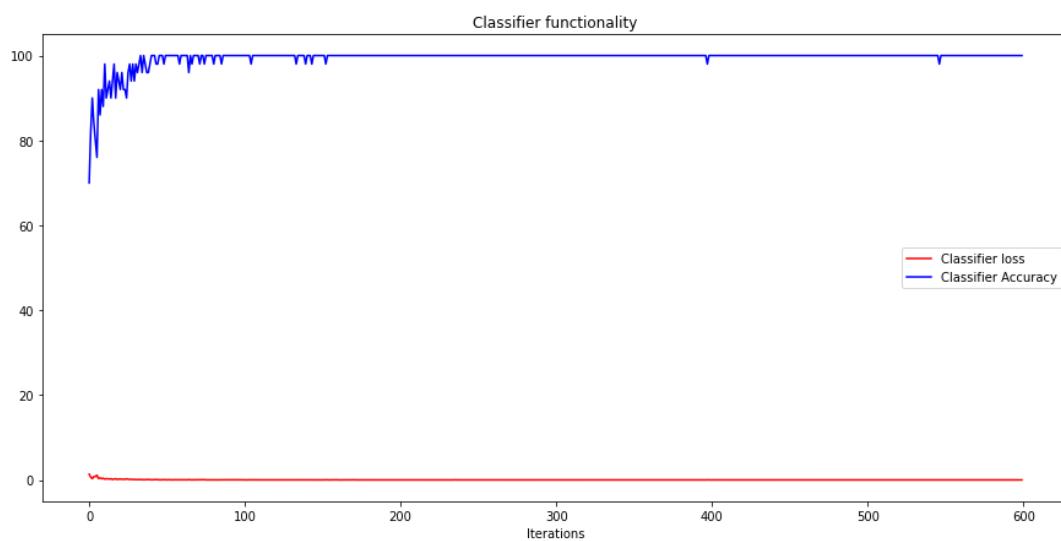
آموزش $VAEGAN$ با ۱۰۰ داده برچسبدار

در ابتدا ۱۰۰ داده‌ی برچسبدار انتخاب کرده و شبکه را بوسیله‌ی آن آموزش می‌دهیم. برای مقایسه با حالات قبل، تا ۶۰۰ ایتریشن آموزش را تکرار می‌کنیم. نمودار خطاهای به شکل زیر است:



شکل ۳۶: نمودار loss برای اجزای مختلف شبکه – آموزش شبکه‌ی $VAEGAN$ با ۱۰۰ داده برچسبدار

و نیز نمودار عملکرد کلسیفایر به شکل زیر است:



شکل ۳۷: عملکرد بخش کلسیفایر – آموزش شبکه‌ی $VAEGAN$ با ۱۰۰ داده برچسبدار

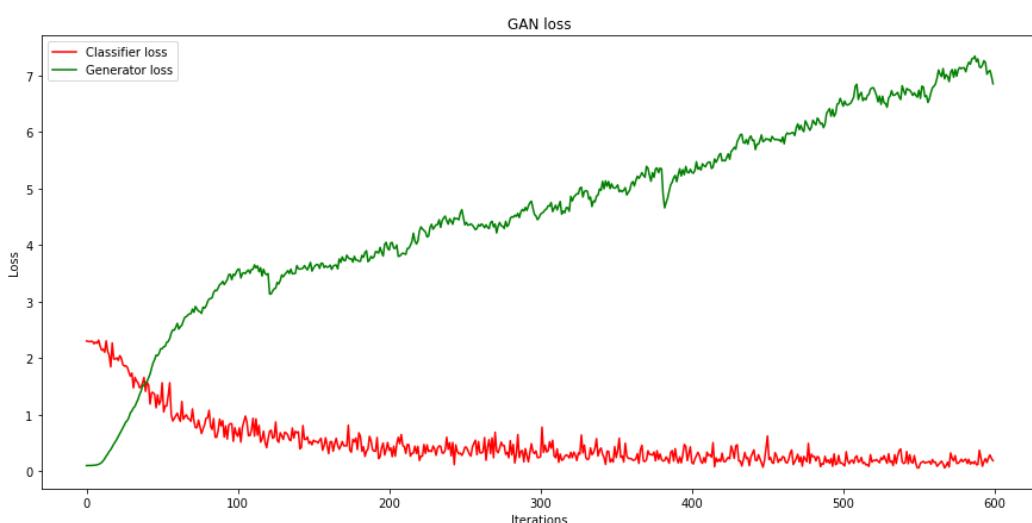
پس از تقریباً ۸۰ ایتریشن، این نمودار اورفیت شده است. همچنین دقت این شبکه بر روی داده‌های تست، ۸۴.۳۵٪ بوده است. نمونه‌ای از تصاویر جنریت شده توسط این شبکه به شکل زیر است:

۳	۶	۷	۵	۱	۵	۹	۸	۳	۹
۷	۸	۹	۶	۷	۸	۳	۶	۴	۶
۸	۶	۹	۴	۶	۴	۴	۹	۷	۴
۶	۳	۷	۴	۳	۴	۹	۳	۱	۳
۹	۵	۲	۳	۹	۳	۱	۲	۴	۶
۶	۷	۹	۷	۸	۹	۳	۸	۴	۲
۳	۳	۷	۸	۱	۹	۲	۸	۷	۹
۱	۳	۳	۱	۵	۴	۲	۶	۵	۰
۷	۷	۳	۴	۹	۸	۶	۳	۸	۴
۹	۷	۹	۳	۹	۹	۶	۷	۵	۰

شکل ۳۸: داده‌های تولید شده توسط شبکه آموزش دیده با ۱۰۰ داده برچسب‌دار – پس از ۶۰۰ ایتریشن

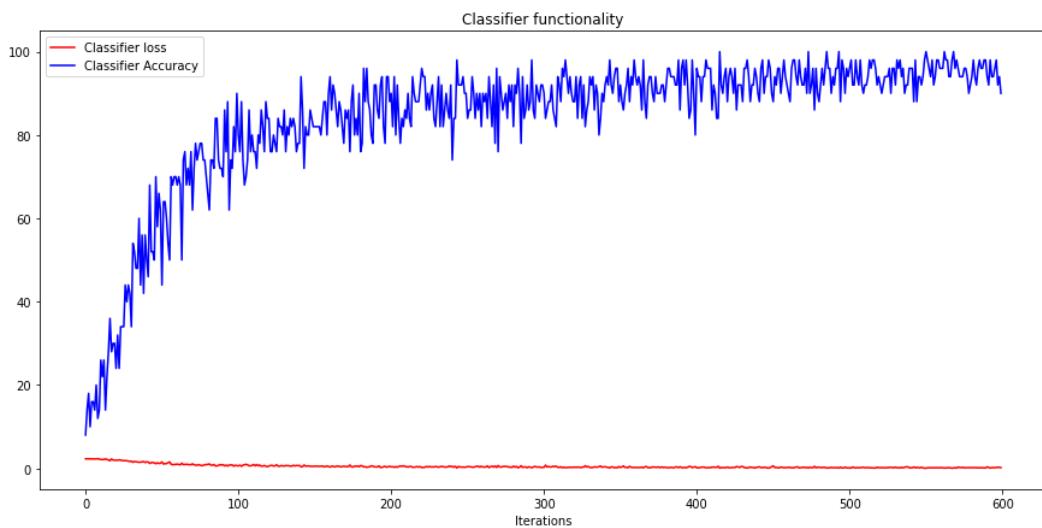
آموزش VAEGAN با ۱۰۰۰ داده برچسب‌دار

در این حالت ۱۰۰۰ داده‌ی برچسب‌دار انتخاب کرده و شبکه را بوسیله‌ی آن آموزش می‌دهیم. برای مقایسه با حالات قبل، تا ۶۰۰ ایتریشن آموزش را تکرار می‌کنیم. نمودار خطاهای به شکل زیر است:



شکل ۳۹: نمودار loss برای اجزای مختلف شبکه – آموزش شبکه‌ی VAEGAN با ۱۰۰۰ داده برچسب‌دار

و نیز نمودار عملکرد کلسیفایر به شکل زیر است:



شکل ۴۰: عملکرد بخش کلسیفایر – آموزش شبکه‌ی **VAEGAN** با ۱۰۰۰ داده برچسبدار

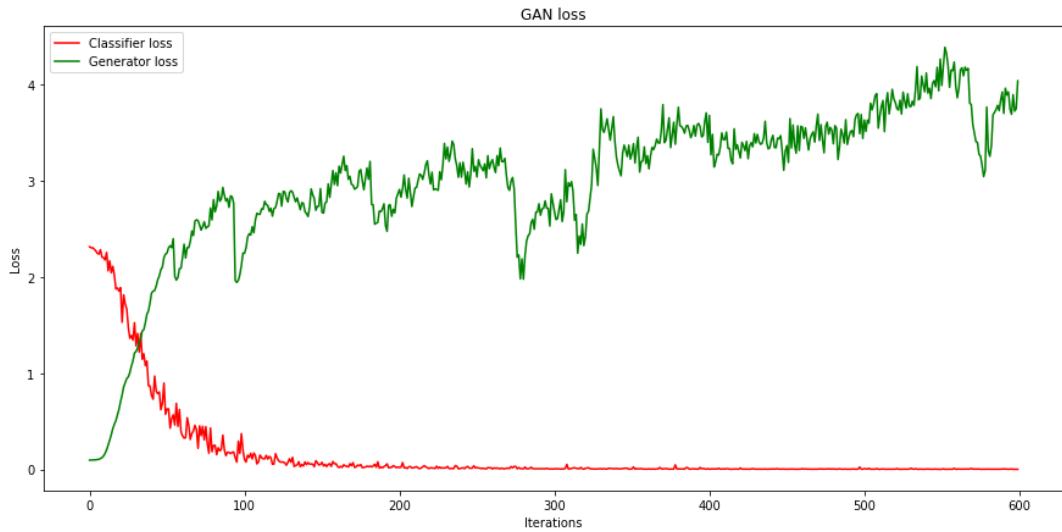
پس از تقریباً ۴۰۰ ایتریشن، این نمودار اورفیت شده است، چرا که تعداد سمپل‌ها بسیار بیشتر بوده است. همچنین دقیق این شبکه بر روی داده‌های تست، ۹۴.۵٪ بوده است. نمونه‌ای از تصاویر جنریت شده توسط این شبکه به شکل زیر است:



شکل ۴۱: داده‌های تولید شده توسط شبکه **VAE GAN** آموزش دیده با ۱۰۰۰ داده برچسبدار – پس از ۶۰۰ ایتریشن

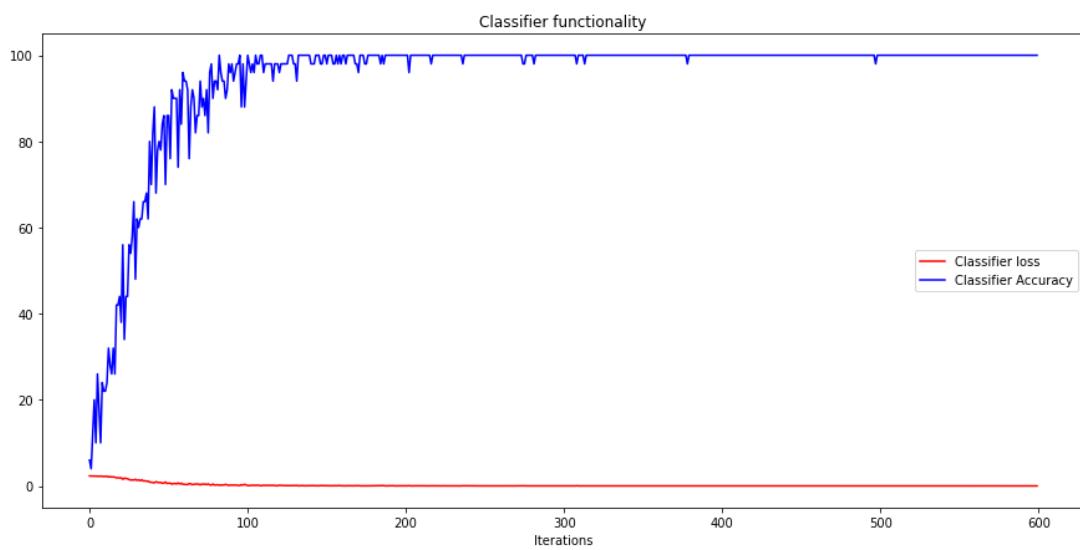
آموزش **VAEGAN** با ۵۰ داده برچسبدار

در این حالت ۵۰ داده‌ی برچسبدار انتخاب کرده و شبکه را بوسیله‌ی آن آموزش می‌دهیم. برای مقایسه با حالات قبل، تا ۶۰۰ ایتریشن آموزش را تکرار می‌کنیم. نمودار خطاهای به شکل زیر است:



شکل ۴۲: نمودار **loss** برای اجزای مختلف شبکه - آموزش شبکه VAEGAN با ۵۰ داده برچسب دار

و نیز نمودار عملکرد کلسیفایر به شکل زیر است:



شکل ۴۳: عملکرد بخش کلسیفایر - آموزش شبکه VAEGAN با ۵۰ داده برچسب دار

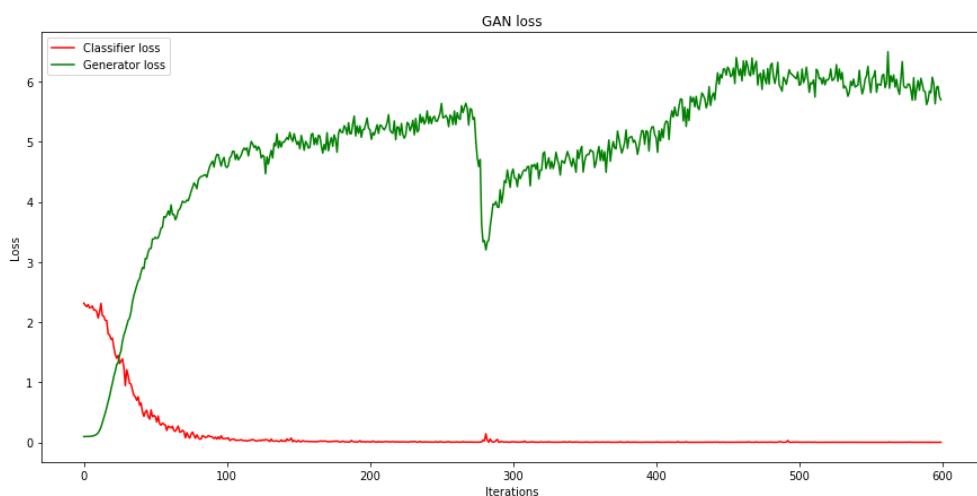
پس از تقریبا ۸۰ ایتریشن، این نمودار اورفیت شده است. همچنین دقت این شبکه بر روی داده های تست، ۷۹.۲٪ بوده است. نمونه ای از تصاویر جنریت شده توسط این شبکه به شکل زیر است:

8	7	2	9	2	4	6	3	7	9
6	0	4	8	9	3	7	2	9	2
5	8	9	3	5	1	5	4	8	4
3	0	8	4	6	6	6	7	8	0
3	9	9	9	5	5	5	7	0	3
9	7	2	3	5	4	7	2	5	8
3	9	2	9	7	6	4	5	3	6
9	4	6	9	6	7	8	5	3	8
4	2	5	5	9	9	4	0	4	0
9	8	7	3	5	4	8	5	4	5

شکل ۴۴: داده‌های تولید شده توسط شبکه VAE GAN آموزش دیده با ۵۰ داده برچسبدار – پس از ۶۰۰ ایتریشن

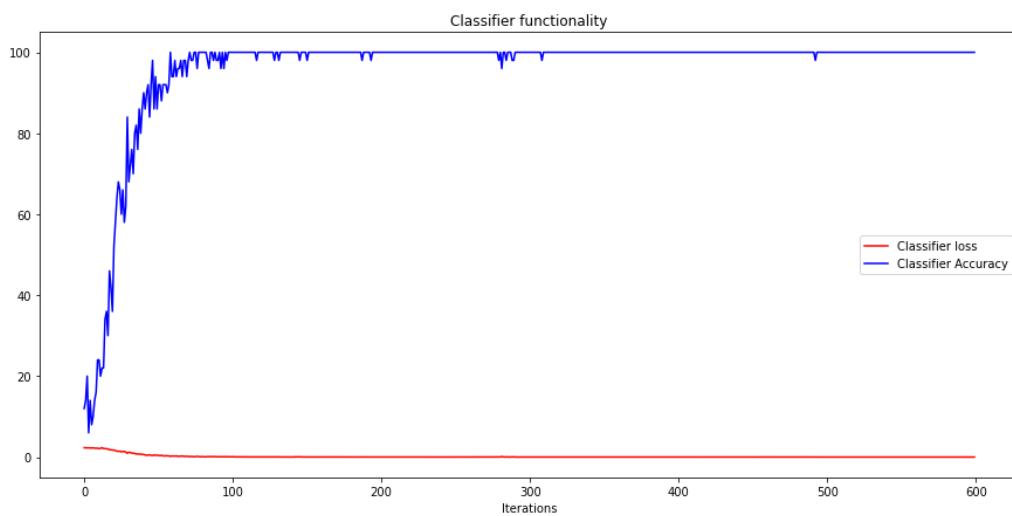
آموزش VAEGAN با ۲۵ داده برچسبدار

در این حالت ۲۵ داده‌ی برچسبدار انتخاب کرده و شبکه را بوسیله‌ی آن آموزش می‌دهیم. برای مقایسه با حالات قبل، تا ۶۰۰ ایتریشن آموزش را تکرار می‌کنیم. نمودار خطاهای به شکل زیر است:



شکل ۴۵: نمودار loss برای اجزای مختلف شبکه – آموزش شبکه‌ی VAEGAN با ۲۵ داده برچسبدار

و نیز نمودار عملکرد کلسیفایر به شکل زیر است:



شکل ۴۶: عملکرد بخش کلسیفایر – آموزش شبکه‌ی VAEGAN با ۲۵ داده برچسبدار

پس از تقریبا ۸۰ ایتریشن، این نمودار اورفیت شده است. همچنین دقت این شبکه بر روی داده‌های تست، ۶۷.۱۳٪ بوده است. نمونه‌ای از تصاویر جنریت شده توسط این شبکه به شکل زیر است:

۰	۳	۳	۹	۴	۷	۵	۹	۸	۳
۴	۴	۰	۶	۳	۷	۹	۸	۹	۲
۵	۰	۶	۸	۴	۸	۶	۷	۱	۹
۷	۴	۶	۳	۴	۶	۵	۳	۳	۹
۳	۶	۳	۹	۷	۳	۸	۷	۱	۴
۷	۷	۰	۳	۸	۴	۶	۳	۶	۶
۹	۳	۸	۰	۷	۰	۹	۹	۶	۸
۶	۴	۰	۸	۵	۵	۹	۰	۳	۳
۱	۳	۹	۷	۱	۳	۳	۹	۰	۳
۷	۶	۹	۷	۰	۵	۵	۹	۸	۶

شکل ۴۷: داده‌های تولید شده توسط شبکه VAE GAN آموزش دیده با ۲۵ داده برچسبدار – پس از ۶۰۰ ایتریشن

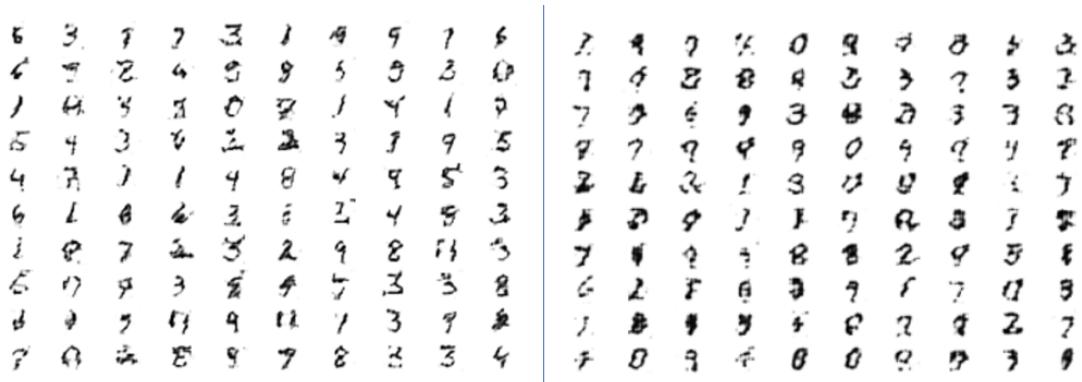
جدول مقایسه

در این بخش، به مقایسه‌ی مدل *VAE-GAN* با دو شبکه‌ی قبلی می‌پردازیم:

(%) <i>VAEGAN accuracy</i>	<i>SGAN accuracy (%)</i>	<i>CNN accuracy (%)</i>	تعداد نمونه‌های برچسب‌دار
۹۴.۵	۹۴.۰۱	۹۱.۹۲	۱۰۰۰
۸۴.۳۵	۸۳.۷۸	۷۸.۷۴	۱۰۰
۷۹.۲	۷۷.۲۲	۷۰.۹۸	۵۰
۶۷.۱۳	۶۵.۷۸	۶۱.۶۱	۲۵

مشاهده می‌شود تقریباً در همه‌ی بخش‌ها، عملکرد شبکه‌ی *VAE GAN* بهتر بوده است.

همچنین برای مقایسه کیفیت تصاویر تولید شده، در دو حالت آموزش بر روی ۱۰۰۰ داده‌ی سهل شده پس از ۶۰۰ ایتریشن را مقایسه می‌کنیم. تصویر سمت راست مربوط به شبکه *SGAN* و تصویر چپ مربوط به شبکه‌ی *VAE GAN* است.



شکل ۴۸: داده‌های تولید شده توسط شبکه *SGAN* آموزش دیده با ۱۰۰۰ داده برچسب‌دار پس از ۶۰۰ ایتریشن – سمت راست داده‌های تولید شده توسط شبکه *VAE GAN* آموزش دیده با ۱۰۰۰ داده برچسب‌دار پس از ۶۰۰ ایتریشن – سمت چپ

تصاویر سمت چپ، اندکی واضح‌تر هستند و این یکی از مواردی است که در مقاله‌ی *VAE GAN* نیز بدان اشاره شده است.

منابع:

- 1- <https://machinelearningmastery.com/semi-supervised-generative-adversarial-network/>
- 2- <https://towardsdatascience.com/vae-variational-autoencoders-how-to-employ-neural-networks-to-generate-new-images-bdeb216ed2c0>
- 3- “Autoencoding beyond pixels using a learned similarity metric” (<https://arxiv.org/pdf/1512.09300.pdf>)

سوال ۲ - DCGAN

الف)

مهم‌ترین هدف در این مقاله، استفاده از شبکه‌های کانولوشنی برای حل مسائل غیر سوپر وایز است. به همین منظور از ساختار شبکه‌های کانولوشنی در یک شبکه عمیق GAN است که هم در generator و هم در discriminator از لایه‌های کانولوشنی برای تولید تصاویر و دسته‌بندی تصاویر استفاده می‌شود.

یکی از مشکلاتی که شبکه‌های GAN با آن‌ها روبرو هستند، ناپایداری در زمان یادگیری است که باعث می‌شود generator خروجی‌های غیر ملموس تولید کند. در این مقاله سعی شده است که ابتدا ساختاری ارائه شود که در آن یادگیری به صورت پایدار صورت بگیرد که این ساختار DCGAN نام‌گذاری شده است. در واقع مهم‌ترین مزیت این شبکه استفاده از لایه‌های کانولوشنی و دی‌کانولوشنی است که موجب می‌شوند تشخیص تصاویر در تفکیک‌کننده و تولید تصاویر در تولیدکننده به نحو بهتری صورت بگیرد.

تا پیش از این مقاله، تلاش‌ها برای اضافه کردن مدل CNN به GAN ناموفق بود. اما در این مقاله سعی شده است که با اعمال تغییراتی استفاده از این مدل ممکن باشد. یکی از این تغییرات این است که به جای استفاده از لایه‌های pooling، از لایه‌های کانولوشنی با stride های بزرگ‌تر از یک استفاده شده است. این موضوع به جنریتور اجازه می‌دهد که خود بتواند داون سمپلینگ مورد نیاز را یاد بگیرد.

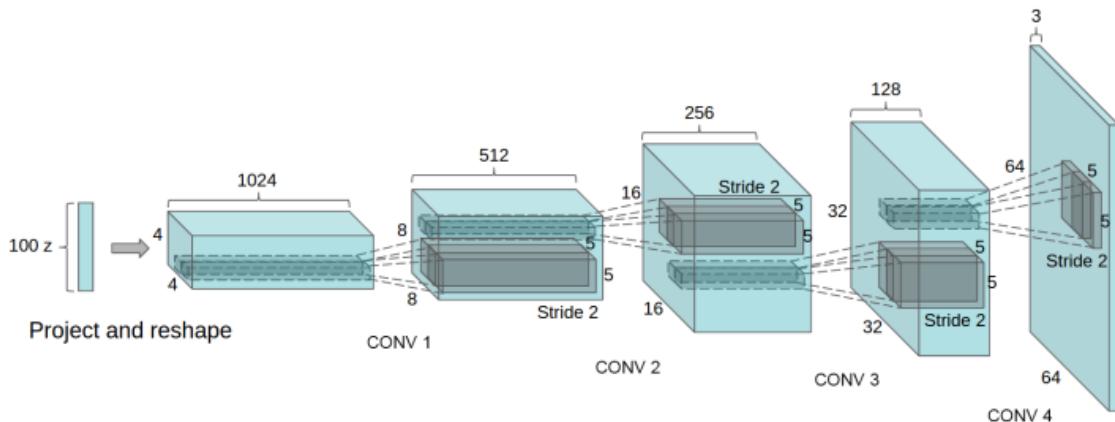
موضوع دوم، استفاده از dropout در میان لایه‌ها بوده است که باعث می‌شود پایداری مدل افزایش پیدا کند.

موضوع سوم، استفاده از Batch Normalization است که با نرمالایز کردن خروجی هر لایه که به لایه بعدی وارد می‌شود، باعث می‌شود پایداری یادگیری افزایش پیدا کند. این موضوع سبب می‌شود که مشکل initialization بد در مدل از بین بود و همچنین جریان گرادیان در مدل‌های عمیق‌تر به نحو بهتری صورت می‌گیرد. همچنین این موضوع سبب می‌شود که مشکل collapse کردن جنریتور به یک نقطه نیز تا حدودی برطرف شود. این موضوع در بخش بعدی بسط بیشتری داده خواهد شد. همچنین باید دقت داشت که در لایه اخر جنریتور و لایه ورودی تفکیک‌کننده نباید نرمالایزیشن صورت بگیرد.

در رابطه با تابع فعال ساز باید گفت که در این مدل از فعال ساز ReLU در لایه‌های هیدن جنریتور و Tanh در لایه اخر استفاده شده است که بین -1 و 1 خروجی تولید می‌کند. لذا لازم است که داده‌های واقعی نیز بین -1 و 1 نرمالایز شوند. همچنین برای لایه‌های هیدن تفکیک‌کننده از LeakyRelu با شبیه ۰.۲ استفاده شده است. در لایه آخر نیز از Sigmoid استفاده شده است.

در آخر باید گفت که به عنوان بهینه‌ساز، از ADAM استفاده شده است. در مقاله توصیه شده است که از مقدار لرنینگ ریت برابر با 2×10^{-4} استفاده شود که مقدار کمتری به نسبت مقدار پیش‌فرض است. همچنین مقدار بتا ۱ نیز برابر با ۰.۵ قرار داده شده است.

ساختار جنریتور مورد استفاده در این مقاله در تصویر زیر قرار گرفته است:



شکل ۴۹ ساختار شبکه جنریتور مورد استفاده در مقاله اصلی (Alec Radford, 2016)

همانطور که دیده می‌شود از کرنل‌هایی با سایز 5×5 استفاده شده است. مقدار stride نیز در لایه‌ها برابر با ۲ است. ورودی اولیه یک بردار به ابعاد ۱۰۰ است که در ابتدا به $1024 \times 4 \times 4$ تغییر شکل پیدا می‌کند. سپس در هر مرحله این عمل $1024 \rightarrow 512 \rightarrow 256 \rightarrow 128$ نصف می‌شود و ابعاد تصویر دو برابر می‌شود. مثلاً در گام دوم عمق ۶۴ می‌شود و ابعاد تصویر 8×8 می‌شود تا در نهایت به یک تصویر 64×64 دست پیدا کنیم که همان ابعاد تصویر واقعی است.

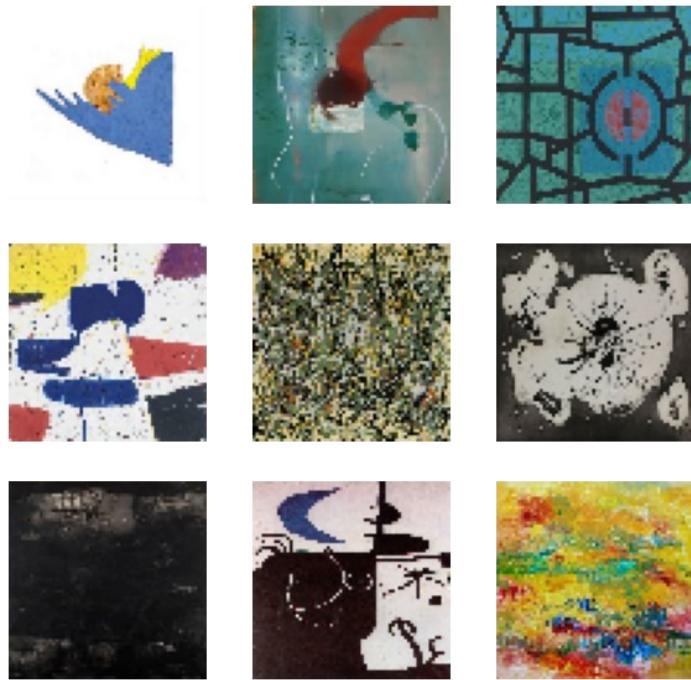
(ب)

در ابتدا لازم است که دیتاست دانلود شود و مورد استفاده قرار گیرد. به همین منظور به طور مستقیم از Kaggle داده‌ها دانلود شدند و در داخل گوگل درایو قرار داده شدند. پس از دانلود تصاویر در گوگل درایو، با استفاده از تکه کد زیر یک کلاس Dataset با استفاده از تصاویر ساخته می‌شود.

```
#Importing data
data_path = "/content/drive/MyDrive/Data_science/abstract-art-gallery/Abstract_gallery/Abstract_gallery"
batch_s = 128
#Import as tf.Dataset
train_ds = tf.keras.preprocessing.image_dataset_from_directory(data_path, label_mode = None, image_size = (64,64), batch_size = batch_s)
```

شکل ۵۰ نحوه ساخت دیتاست از تصاویر در keras

برای درک بهتر تصاویر دیتاست، چند تصویر به صورت تصادفی رسم می‌شود که در شکل ۵۱ نمونه‌ای از آن دیده می‌شود.



شکل ۵۱ چند تصویر نمونه از دیتاست AbstractArt

تنها پیش‌پردازشی که لازم است صورت بگیرد، نرم‌الایز کردن داده‌ها است. از آنجایی که خروجی جنریتور \tanh در نظر گرفته شده است، پس داده‌ها را باید به بازه -1 تا 1 مپ کنیم. پس از آن لازم است که ساختار تولید‌کننده و تفکیک‌کننده را در keras تعریف کنیم. ما در اینجا دقیقاً از همان مشخصات گفته شده در مقاله استفاده کردایم و در تولید‌کننده نیز از فعال‌ساز Relu استفاده کردایم. ساختار در نظر گرفته شده برای تولید‌کننده در تصویر زیر قرار گرفته است.

None Model: "sequential_1"		
Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 16384)	1638400
batch_normalization_4 (BatchNormalization)	(None, 16384)	65536
relu (ReLU)	(None, 16384)	0
reshape (Reshape)	(None, 4, 4, 1024)	0
conv2d_transpose (Conv2DTranspose)	(None, 8, 8, 512)	13107200
batch_normalization_5 (BatchNormalization)	(None, 8, 8, 512)	2048
relu_1 (ReLU)	(None, 8, 8, 512)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 16, 16, 256)	3276800
batch_normalization_6 (BatchNormalization)	(None, 16, 16, 256)	1024
relu_2 (ReLU)	(None, 16, 16, 256)	0
conv2d_transpose_2 (Conv2DTranspose)	(None, 32, 32, 128)	819200
batch_normalization_7 (BatchNormalization)	(None, 32, 32, 128)	512
relu_3 (ReLU)	(None, 32, 32, 128)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 64, 64, 3)	9600
activation (Activation)	(None, 64, 64, 3)	0

شکل ۵۲ ساختار تولیدکننده

و ساختار تفکیک‌کننده نیز در تصویر زیر قرار گرفته است. لازم به ذکر است که فعال‌ساز لایه آخر، sigmoid است. همچنین تعداد لایه‌های کانولوشنی ابتدا از ۶۴ آغاز می‌شود و سپس در نهایت به ۵۱۲ لایه می‌رسد.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 64)	4864
batch_normalization (BatchNormalization)	(None, 32, 32, 64)	256
leaky_re_lu (LeakyReLU)	(None, 32, 32, 64)	0
dropout (Dropout)	(None, 32, 32, 64)	0
conv2d_1 (Conv2D)	(None, 16, 16, 128)	204928
batch_normalization_1 (BatchNormalization)	(None, 16, 16, 128)	512
leaky_re_lu_1 (LeakyReLU)	(None, 16, 16, 128)	0
dropout_1 (Dropout)	(None, 16, 16, 128)	0
conv2d_2 (Conv2D)	(None, 8, 8, 256)	819456
batch_normalization_2 (BatchNormalization)	(None, 8, 8, 256)	1024
leaky_re_lu_2 (LeakyReLU)	(None, 8, 8, 256)	0
dropout_2 (Dropout)	(None, 8, 8, 256)	0
conv2d_3 (Conv2D)	(None, 8, 8, 512)	3277312
batch_normalization_3 (BatchNormalization)	(None, 8, 8, 512)	2048
leaky_re_lu_3 (LeakyReLU)	(None, 8, 8, 512)	0
dropout_3 (Dropout)	(None, 8, 8, 512)	0
flatten (Flatten)	(None, 32768)	0
dense (Dense)	(None, 1)	32769

شکل ۵۳ ساختار تفکیک‌کننده

پس از تعریف شبکه‌های تولیدکننده و تفکیک‌کننده، یک کلاس به نام GAN توسعه داده شد که در آن فرآیند یادگیری برای این شبکه صورت می‌گیرد. در ابتدا لازم است که تابع بهینه ساز و سایز batch مورد استفاده مورد بررسی قرار گیرند. در این مسئله نیز از تابع بهینه ساز Adam با پارامترهای ذکر شده در مقاله (نرخ یادگیری ۰.۰۰۰۲ و مقدار بتا ۱ برابر با ۰.۵ برای هردو شبکه استفاده می‌شود).

پس از این قسمت لازم است که مقدار loss برای تفکیک‌کننده و تولیدکننده مشخص شود. از آنجا که خروجی تفکیک‌کننده sigmoid است، پس باید لیبل‌ها را برابر با ۱ و ۰ قرار دهیم. به همین منظور مطابق با مقاله اصلی، از cross_entropy به عنوان تابع هزینه استفاده می‌کنیم. همچنین لیبل داده‌های واقعی برای تفکیک کننده برابر با ۱ و برای داده‌های فیک برابر با ۰ در نظر گرفته شده است. همچنین برای تولیدکننده نیز لیبل داده‌های فیک برابر با ۱ در نظر گرفته شده است. درواقع تلاش تولیدکننده باید تولید

تصاویری باشد که باعث شود تفکیک‌کننده آن را جزو تصاویر واقعی تفکیک کنند. این دوتابع هزینه در تصویر زیر قابل مشاهده هستند.

```
def discriminator_loss(self, real_output, fake_output):
    real_loss = self.cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = self.cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(self, fake_output):
    return self.cross_entropy(tf.ones_like(fake_output), fake_output)
```

شکل ۵۴ تابع هزینه مورد استفاده

در نهایت لازم است که گام یادگیری برای هر batch در نظر گرفته شود. این بخش از آموزش قرار گرفته در وبسایت تنسورفلو الگوبرداری شده است.^۱

```
@tf.function # Compiles a function into a callable TensorFlow graph
def train_step(self, images):
    noise = tf.random.normal([self.batch_size, self.noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = self.generator(noise, training=True)

        real_output = self.discriminator(images, training=True)
        fake_output = self.discriminator(generated_images, training=True)

        gen_loss = self.generator_loss(fake_output)
        disc_loss = self.discriminator_loss(real_output, fake_output)

        gradients_of_generator = gen_tape.gradient(gen_loss, self.generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(disc_loss, self.discriminator.trainable_variables)

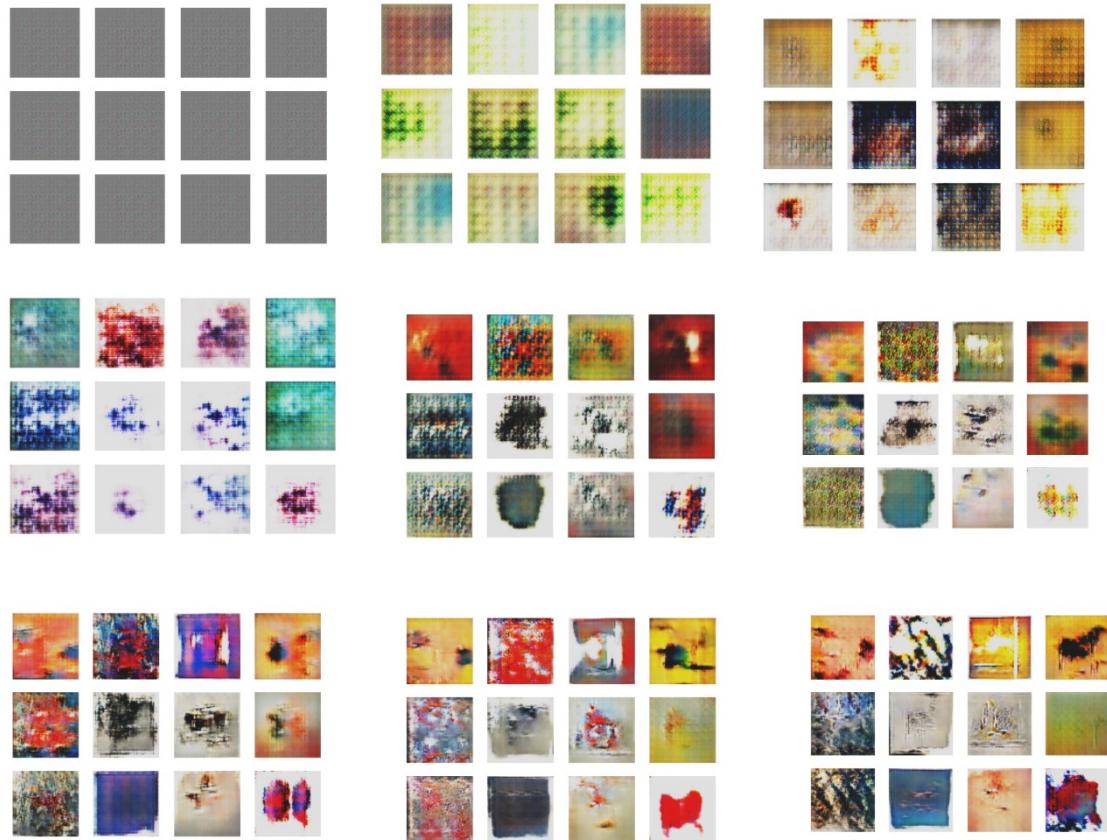
        self.generator_optimizer.apply_gradients(zip(gradients_of_generator, self.generator.trainable_variables))
        self.discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, self.discriminator.trainable_variables))

    return gen_loss, disc_loss
```

شکل ۵۵ حلقه یادگیری

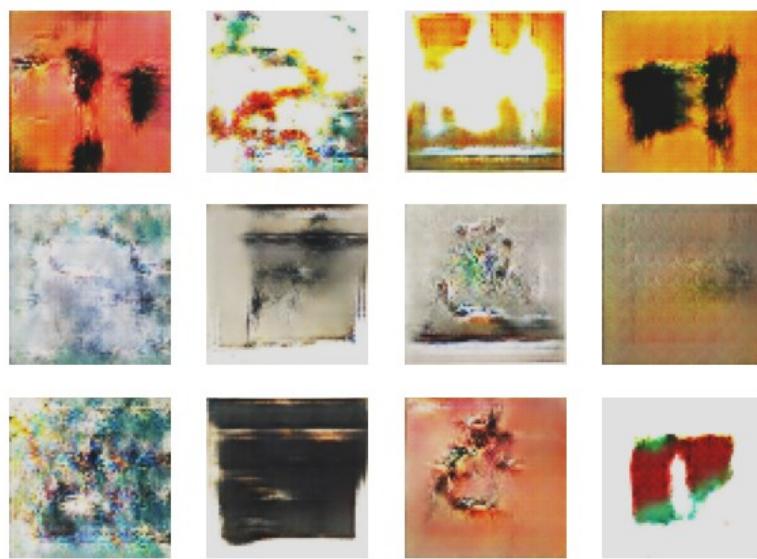
همانطور که دیده می‌شود ابتدا یک نویز به ابعاد batch تولید می‌شود. سپس با استفاده از این نویز تصاویر فیک تولید می‌شوند و بعد از آن با استفاده از یک batch داده‌های واقعی، مقدار لاس برای تولیدکننده و تفکیک‌کننده به دست می‌آید. پس از آن مقدار گرادیان‌ها محاسبه می‌شود و بر اساس آن با استفاده از تابع بهینه‌ساز، وزن‌ها اصلاح می‌شوند.

با در نظر گرفتن `batch_size` برابر با ۱۲۸ و برای ۴۵۰ ایپاک، نتایج زیر در هر ۵۰ ایپاک به دست آمد. (از چپ به راست)



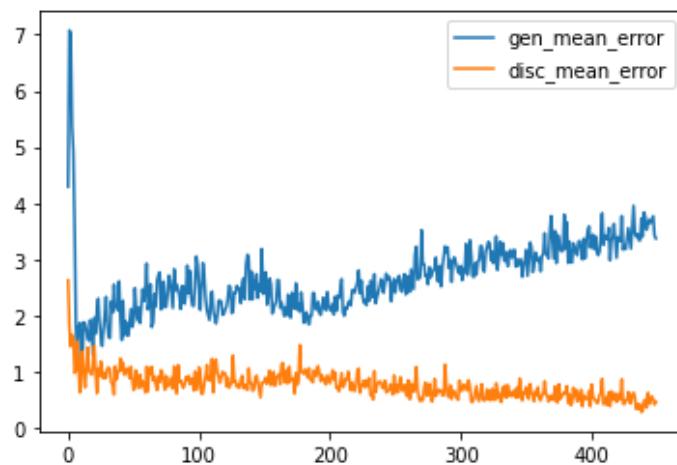
شکل ۵۶ خروجی شبکه برای ۱۲ ورودی نویز ثابت در طول یادگیری

و درنهایت در ایپاک ۴۵۰، خروجی زیر برای شبکه به دست آمد.



شکل ۵۷ نمونه‌ای از خروجی شبکه در ایپاک ۴۵۰

هرچند نمودار خطای شبکه‌های Gan لروما نشانگر خوبی برای همگرایی شبکه یا تولید نمونه‌های خوب توسط تولیدکننده نیستند، اما برای درک بهتر عملکرد شبکه می‌تواند مورد استفاده قرار گیرد. در زیر نمودار خطای تولیدکننده و تفکیک‌کننده قرار گرفته است.



شکل ۵۸ نمودار خطای تولیدکننده و تفکیک کننده

همانطور که دیده می‌شود در انتهای مقداری خطای تولیدکننده زیاد شده است. اما این موضوع به قوی‌تر شدن تفکیک کننده باز می‌گردد و همانطور که دیده شد، تصاویر خروجی کیفیت بهتری پیدا کرده‌اند و بیشتر به تصاویر واقعی شبیه شده‌اند.

(ج)

مشکل عدم همگرایی

یکی از مهم‌ترین مشکلات شبکه‌های GAN، عدم توانایی آن‌ها در همگرایی است. این مشکل در چندین مقاله مورد بررسی قرار گرفته است. به عنوان مثال یکی از موارد می‌تواند این باشد که تفکیک‌کننده در ابتدا بسیار قوی باشد و خروجی آن به یک خروجی step نزدیک شود. در این صورت تولیدکننده به خوبی نمی‌تواند همگرا شود. همچنین با ران‌های بسیار زیادی که گرفتیم، متوجه شدیم که شبکه GAN بسیار به هایپرپارامترها حساس است و این موضوع می‌تواند توانایی شبکه را در همگرایی تحت تاثیر بگذارد. دو راهکار نیز برای این مشکل توصیه شده است.

راهکار اول، افزودن نویز پیوسته در لایه‌های ابتدایی تفکیک‌کننده است. این موضوع مشابه با اعمال قید بر روی وزن‌های این شبکه است و باعث ضعیفتر شدن تفکیک‌کننده در ابتدا می‌شود و همچنین ورودی‌های شبکه را به یک توزیع پیوسته نزدیک‌تر می‌سازد [1].

راهکار دوم نیز افزودن پنالتی به ازای وزن‌های شبکه است [2]. نحوه انجام این کار نیز این است که یک ترم به تابع هزینه افزوده می‌شود که در آن بر اساس نرم گرادیان‌ها پنالتی داده می‌شود.

مشکل محو شدن گرادیان و mode collapse

در ابتدا لازم است که در رابطه با این دو مشکل توضیحاتی را ارائه دهیم. مشکل اول محو شدن گرادیان است. در یکی از تحقیقات [1] نشان داده شده است که اگر تفکیک‌کننده بیش از اندازه قوی شود، در این صورت جنریتور می‌تواند به دلیل پدیده محو شدن گرادیان، به خوبی دیگر نتواند عمل کند. این موضوع تا حدودی در ران گرفته شده از شبکه در بالا نیز دیده می‌شود. پدیده محو شدن گرادیان به طور کلی می‌تواند در تمامی شبکه‌های عمیق نیز اتفاق افتد که اگر از یک جایی به بعد گرادیان‌ها کوچک شوند، دیگر تاثیر آن در لایه‌های بالاتر دیده نخواهد شد. یکی از راهکارها استفاده از batch_normalization است که در این شبکه ما نیز مورد استفاده قرار گرفته است. راهکار بعدی استفاده از یک تابع هزینه جدید است که تحت عنوان Wasserstein loss شناخته می‌شود [3]. راهکار دیگر نیز استفاده از تابع هزینه اصلاح شده minmax است که ما از همان راه اول استفاده خواهیم کرد.

مشکل دوم، Mode Collapse است. مطلوب ما معمولاً این است که شبکه بتواند بازه متنوعی از تصاویر را که مشابه با تصاویر مختلفی از شبکه اصلی است را تولید کند. اما مشکل زمانی ایجاد می‌شود که تولیدکننده یادمیگیرد که یک نوع از تصویر مطلوب تفکیک‌کننده است. در این صورت ممکن است که تولیدکننده از یک جایی به بعد تنها سعی کند که همان نوع از تصویر را تولید کند و درواقع سعی می‌کند به ازای تمامی

ورودی‌ها، یک نوع تصویر یکسان را تولید کند. قاعده‌تا در این زمان بهترین استراتژی برای تفکیک‌کننده این است که اگر تولیدکننده دائماً یک نوع تصویر را تولید می‌کند، این تصاویر را به عنوان داده فیک لیل‌گذاری کند. اما مشکل زمانی ایجاد می‌شود که تفکیک‌کننده در یک لوکال مینیمم گیر کرده باشد و نتواند خارج شود، در این صورت جنریتور می‌تواند تصویر مطلوب این تفکیک‌کننده گیر کرده در لوکال مینیمم را پیدا کرده و دائماً همان تصویر را تولید کند. این مشکل mode collapse نامیده می‌شود. دو راهکار برای این مشکل پیشنهاد می‌شود.

یک راه حل استفاده از Wasserstein loss است که می‌تواند مشکل محو شدن گرادیان را برطرف سازد و با این راهکار، امکان گیر کردن در یک لوکال مینیمم کاهش پیدا می‌کند.

راهکار دوم استفاده از Unrolled GAN است [4]. در این شبکه لاس جنریتور علاوه بر در نظر گرفتن دسته‌بندی‌های تفکیک‌کننده فعلی، دسته‌بندی‌های تفکیک‌کننده‌های آینده را نیز در نظر می‌گیرد. لذا در چنین حالتی امکان اینکه جنریتور برای یک نوع تفکیک‌کننده خاص بهینه شود کاهش می‌آید. ما برای حل این مشکل نیز از راه اول استفاده می‌کنیم که درواقع دو مشکل را برطرف می‌سازد.

استفاده از Wasserstein loss

طبق مقاله WGAN به جای استفاده از خروجی sigmoid در لایه آخر تفکیک‌کننده، باید از یک خروجی خطی استفاده شود. همچنین به جای استفاده از تابع هزینه cross entropy، از میانگین خروجی‌ها به عنوان هزینه استفاده می‌شود. نحوه کار نیز به این نحو است که برای تفکیک‌کننده، لیبل -1 برای داده‌های واقعی و +1 برای داده‌های فیک در نظر گرفته می‌شود. در این صورت تفکیک‌کننده همیشه سعی خواهد کرد که مقدار تابع هزینه منفی‌تر شود. برای تولیدکننده نیز مقدار لیبل -1 برای خروجی‌هاییش در نظر گرفته می‌شود و بنابراین همیشه سعی خواهد کرد که مقدار خروجی منفی‌تر شود. توابع هزینه اصلاح شده به شرح زیر هستند:

```
def discriminator_loss(self, real_output, fake_output):
    real_loss = tf.reduce_mean(real_output)
    fake_loss = tf.reduce_mean(fake_output)
    return (fake_loss - real_loss)

def generator_loss(self, fake_output):
    return -tf.reduce_mean(fake_output)
```

شکل ۵۹ توابع هزینه اصلاح شده WGAN

اما اصلاحات به همینجا ختم نمی‌شوند. در مقاله اصلی توضیح داده شده است که استفاده از توابع بهینه‌سازی که در آن‌ها از مومنتوم استفاده می‌شود ممکن است باعث شود که همگرایی در شبکه رخ ندهد. بنابراین به جای استفاده از تابع بهینه ساز Adam، از تابع RMSprop استفاده می‌شود. این تابع بهینه ساز به همراه پارامترهای آن در تصویر زیر دیده می‌شود.

```
self.generator_optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.00005)
self.discriminator_optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.00005)
```

شکل ۶۰ تابع بهینه WGAN

همانطور که دیده می‌شود از نرخ یادگیری کوچکتری نیز استفاده شده است.

تغییر بعدی clip کردن وزن‌های شبکه است. موضوعی که به خصوص در شبکه‌های عمیق‌تر دیده می‌شود، بزرگ‌شدن بیش از حد وزن‌های تفکیک‌کننده است که می‌تواند منجر به تولید داده‌های بسیار بزرگ شود و همگرایی را تحت تاثیر قرار می‌دهد. برای اعمال قید بر روی وزن‌های شبکه، از تکه کد زیر استفاده شده است.

```
from keras.constraints import Constraint
from keras.initializers import RandomNormal

class ClipConstraint(Constraint):
    def __init__(self, clip_value):
        self.clip_value = clip_value

    def __call__(self, weights):
        return backend.clip(weights, -self.clip_value, self.clip_value)

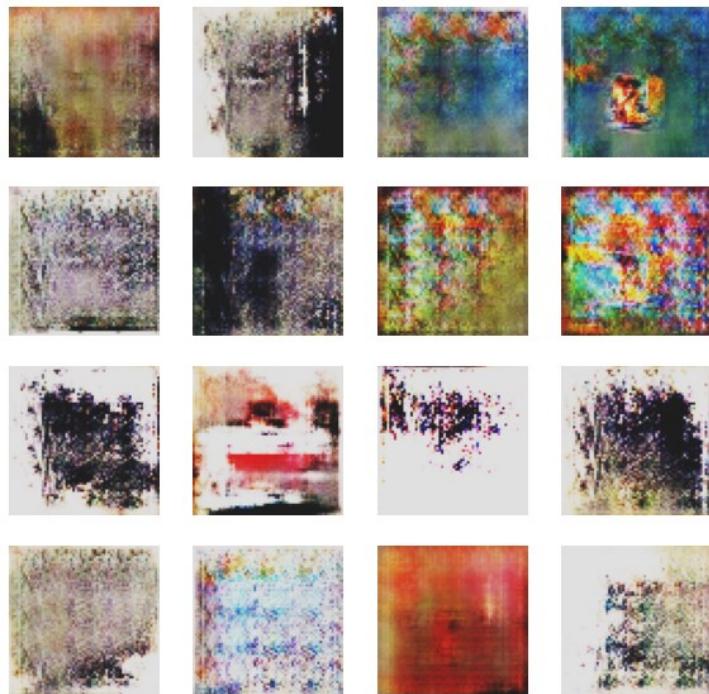
    def get_config(self):
        return {'clip_value': self.clip_value}
```

شکل ۶۱ نحوه محدود کردن وزن‌های شبکه

که در اینجا وزن‌های شبکه بین ۰.۰۱ و ۰.۰۰۱ محدود می‌شوند. طبق مقاله گفته شده است که می‌توان batch_normalization را نیز حذف کرد. ما در هردو حالت شبکه را امتحان کردیم و دیده شد که در حالت عدم وجود لایه نرمالایزیشن، وزن‌های شبکه بیش از حد بزرگ می‌شوند و همگرایی رخ نمی‌دهد. موضوع آخری که در مقاله توصیه شده است، یادگیری بیشتر برای تفکیک‌کننده است. ما در این شبکه به ازای هر سه یادگیری تفکیک‌کننده بر روی batch‌ها، یک بار نیز تولیدکننده را یاد می‌دهیم. در واقع در

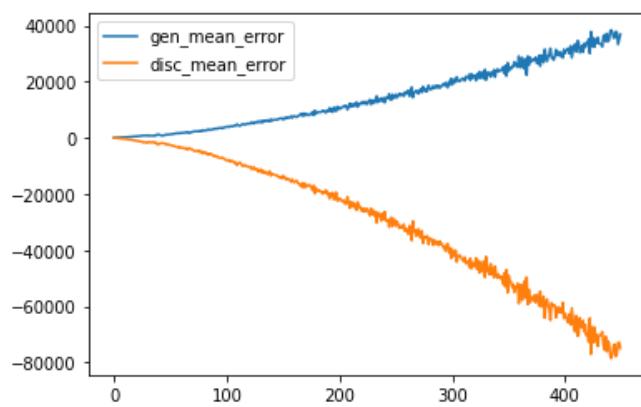
اینجا سعی می‌شود که تفکیک‌کننده قوی‌تری تولید شود. البته در مقاله توصیه شده بود که این نسبت برابر با ۵ باشد.

خروجی شبکه با استفاده از پارامترهای گفته شده، پس از ۴۵۰ ایپاک به شکل زیر به دست آمد:



شکل ۶۲ خروجی شبکه WGAN با پارامترهای دیفالت مقاله پس از ۴۵۰ ایپاک

همانطور که دیده می‌شود، همگرایی شبکه به خوبی صورت نگرفته است و داده‌ها هنوز فیک به نظر می‌رسند. با بررسی مقدار تابع هزینه متوجه شدیم که واگرایی زیادی رخ داده است.



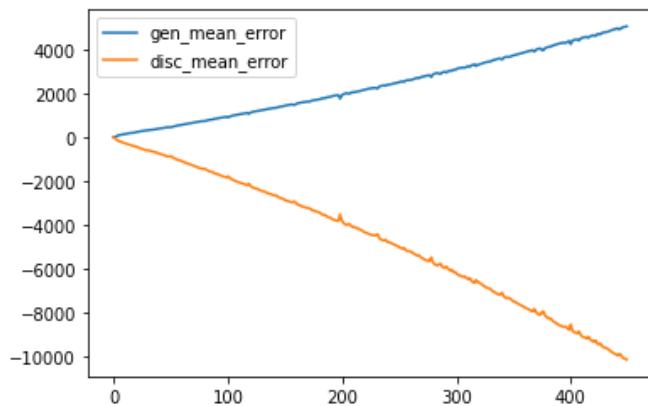
شکل ۶۳ تابع هزینه WGAN با پارامترهای دیفالت

ابتدا تصور شد که علت این واگرایی، بزرگ بودن مقدار نرخ یادگیری به ۱۰۰۰۰۰۰، خروجی زیر برای شبکه پس از ۴۵۰ ایپاک به دست آمد.



شکل ۶۴ خروجی شبکه WGAN با نرخ یادگیری کوچکتر

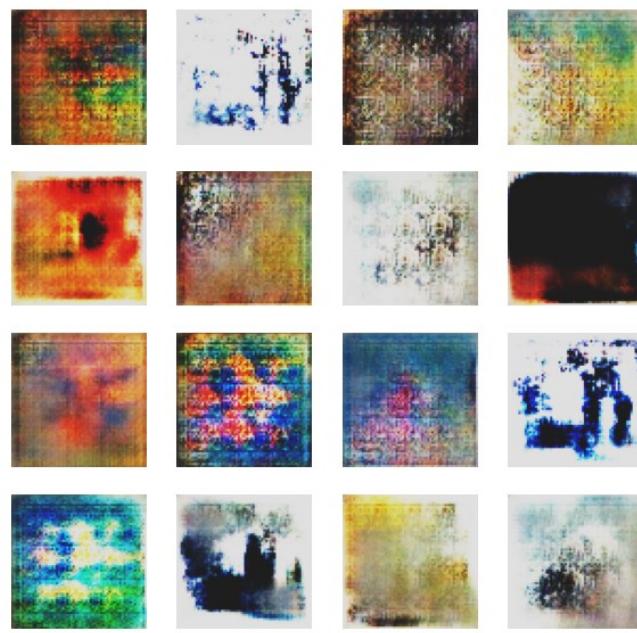
همانطور که دیده می‌شود نه تنها مشکل برطرف نشده است، بلکه حالتی مشابه با mode collapse نیز در حال رخ دادن است! بنابراین سعی کردیم با راهکار ارائه شده برای حل مشکل همگرایی مشکل را برطرف کنیم. مقدار هزینه نیز در نمودار زیر دیده می‌شود.



شکل ۶۵ مقدار تابع هزینه با کاهش نرخ یادگیری

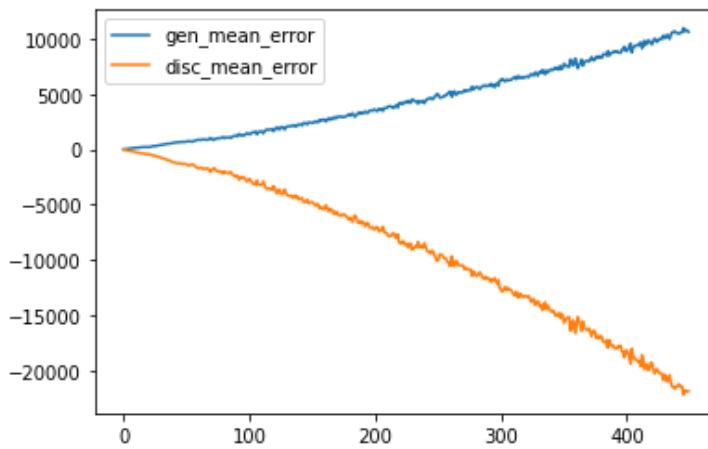
رفع مشکل همگرایی

همانطور که دیده شد در شبکه WGAN عدم همگرایی زیادی رخ داده است. همانطور که گفته شد یکی از راهکارها استفاده از نویز در لایه‌های تفکیک‌کننده است. به همین منظور یک نویز گاوی با واریانس ۰.۱ به تفکیک‌کننده اضافه شد. خروجی شبکه پس از ۴۵۰ ایپاک به شکل زیر به دست آمد.



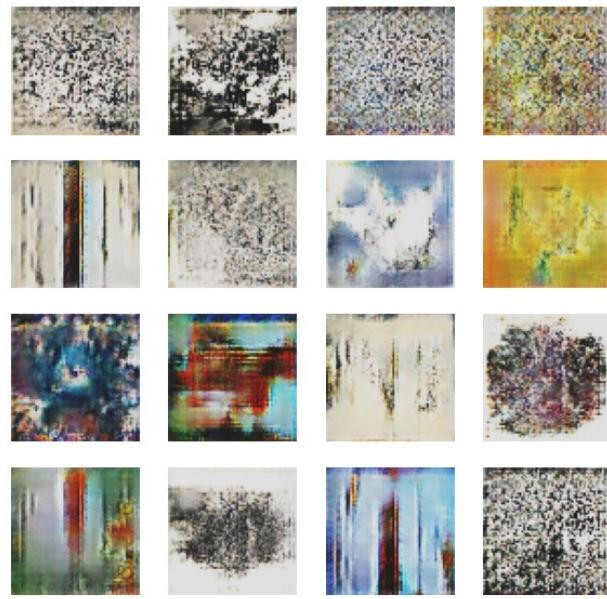
شکل ۶۶ خروجی شبکه WGAN با افزودن نویز به تفکیک‌کننده

همانطور که دیده می‌شود خروجی‌های بهتری تولید شده‌اند و تنوع مناسب‌تری هم دارند اما مشکل این است که همچنان همگرایی می‌تواند عملکرد بهتری داشته باشد. هرچند در نمودار زیر مقدار لاس کمتری به نسبت حالت بدون نویز دیده می‌شود، اما همچنان عدم همگرایی در شبکه وجود دارد. هرچند شدت آن کمتر است و نتایج بهتری نیز تولید شده است.



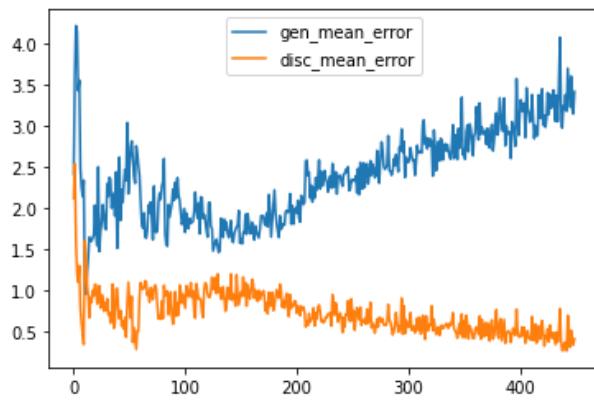
شکل ۶۷ تابع هزینه WGAN با افزودن نویز

همچنین به سراغ شبکه اولیه نیز رفتیم و به لایه‌های آن نیز نویز اضافه کردیم. هرچند در آن شبکه چندان مشکل همگرایی جدی نبود. خروجی زیر برای شبکه DCGAN اولیه با افزودن نویز به دست آمد.



شکل ۶۸ خروجی شبکه DCGAN اولیه با افزودن نویز به تفکیک‌کننده

و مقدار تابه هزینه نیز در نمودار زیر دیده می‌شود.



شکل ۶۹ مقدار تابع هزینه DCGAN اولیه با افزودن نویز به تفکیک‌کننده

همانطور که دیده می‌شود هرچند مقداری نتیجه در میانه بهتر شده است، اما در نهایت عدم همگرایی نیز دیده می‌شود. البته شاید پس از مدتی و طی ایپاک‌های بیشتر، این مشکل برطرف می‌شد.

رفع مشکل (WGAN-GP) WGAN

در این مرحله ما از نتایج به دست آمده برای WGAN راضی نبودیم. به همین منظور با بررسی مقالات متوجه شدیم که در یکی از مقالات موضوع عدم همگرایی WGAN مورد بررسی قرار گرفته است[5]. در این مقاله گفته می‌شود که Clip کردن وزن‌ها ایده مناسبی به خصوص در شبکه‌های بزرگ نمی‌تواند باشد و می‌تواند به عدم همگرایی شبکه منجر شود. به همین منظور به جای این روش توصیه شده است که با در نظر گرفتن پنالتی برای وزن‌های تفکیک‌کننده، این مشکل برطرف شود. به همین منظور ما تکه کد

مریبوط به Clip کردن وزن‌ها را حذف کردیم و به جای آن تکه کد مربوط به پنالتی دادن به وزن‌ها افزوده شد که در تصویر زیر این موضوع دیده می‌شود. این تکه کد از این آموزش برگرفته شده است^۱. در ضمن این کار عملکردی مشابه با افزودن نویز را در شبکه ایجاد می‌کند که می‌تواند موجب همگرایی بهتر شود.

```
def gradient_penalty(self, real_images, fake_images):
    """Calculates the gradient penalty.

    This loss is calculated on an interpolated image
    and added to the discriminator loss.
    """

    # Get the interpolated image
    alpha = tf.random.normal([real_images.shape[0], 1, 1, 1], 0.0, 1.0)
    diff = fake_images - real_images
    interpolated = real_images + alpha * diff

    with tf.GradientTape() as gp_tape:
        gp_tape.watch(interpolated)
        # 1. Get the discriminator output for this interpolated image.
        pred = self.discriminator(interpolated, training=True)

        # 2. Calculate the gradients w.r.t to this interpolated image.
        grads = gp_tape.gradient(pred, [interpolated])[0]
        # 3. Calculate the norm of the gradients.
        norm = tf.sqrt(tf.reduce_sum(tf.square(grads), axis=[1, 2, 3]))
        gp = tf.reduce_mean((norm - 1.0) ** 2)
    return gp
```

شکل ۷۰ تکه کد پنالتی دادن به گرادیان‌ها

و به این شکل به هزینه محاسبه شده در WGAN معمولی افزوده می‌شود. در مقاله توصیه شده است که این تکه کد `gp_weight` یا لاندا برابر با ۱۰ انتخاب شود. با این حال ما مقادیری مثل ۲، ۵ و ۱۰ را نیز امتحان کردیم.

```
with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
    generated_images = self.generator(noise, training=train_generator_state)

    real_output = self.discriminator(images, training=True)
    fake_output = self.discriminator(generated_images, training=True)

    gen_loss = self.generator_loss(fake_output)
    disc_loss = self.discriminator_loss(real_output, fake_output)

    gp = self.gradient_penalty(images, generated_images)

    disc_loss = disc_loss + gp * self.gp_weight
```

شکل ۷۱ تابع هزینه اصلاح شده

همچنین در مقاله گفته شده است که می‌توان به جای ADAM از RMSprop استفاده کرد و نرخ یادگیری را نیز بزرگتر قرار داد.

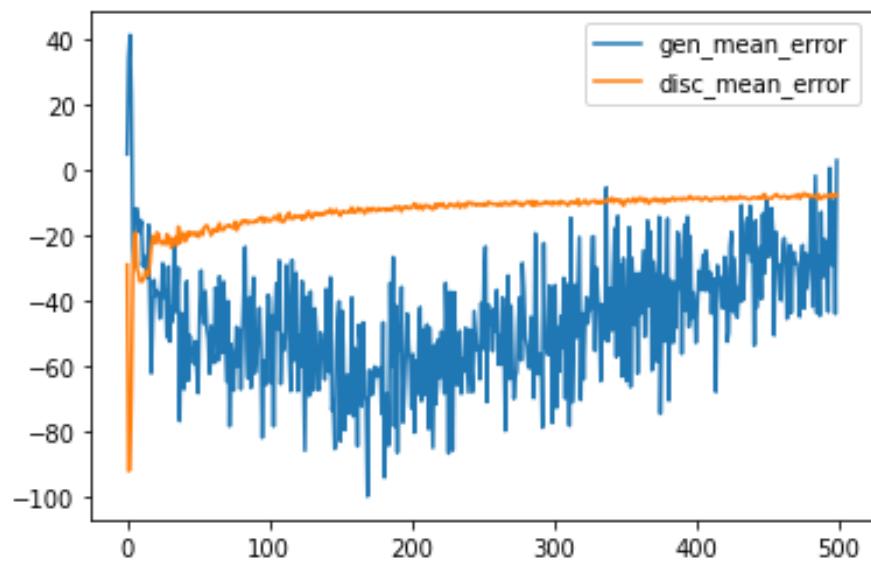
در ضمن لایه‌های نرمالیزیشن نیز باید حذف شوند و مقدار دارپ اوت را نیز می‌توان کوچکتر قرار داد. ما در اینجا آزمون و خطاهای بسیار زیادی (بیش از ۵-۶ بار تغییر هایپرپارامترها) را انجام دادیم تا بتوانیم مقدار بهینه پارامترها را برای حالات مختلف پیدا کنیم. مشکلی که بسیار دیده شد، نوسانات زیاد مدل بود. به نحوی که جنریتور ابتدا یک پترن را پیدا می‌کرد که در آن عکس‌ها لاس پایینی را ایجاد می‌کردند. سپس دیسکریمینیتور بعد از چند ایپاک این تصاویر را رد می‌کرد و دوباره لاس بالا میرفت و این چرخه همیشه ادامه داشت و هرچند شبکه واگرا نمیشد ولی همگرا هم به خوبی نمیشد و پس از چند ایپاک نوسان داشت و مدل جدیدی خروجی میداد. پس از چند آزمون و خطا، به این نتیجه رسیدیم که برای دیسکریمینیتور اگر از تابع بهینه ساز RMSprop استفاده کنیم و برای تولیدکننده تابع بهینه‌ساز Adam به نتیجه بهتری دست پیدا می‌کنیم. همچنین لیبل تصاویر فیک و واقعی را تغییر دادیم و برای تصاویر واقعی برابر با ۱ و برای تصاویر فیک -۱ را در نظر گرفتیم (هرچند این موضوع چندان تاثیر گذار نیست و تفاوتی ندارد). خروجی شبکه پس از ۵۰۰ ایپاک به شرح زیر است:



شکل ۷۲ خروجی شبکه WGAN-GP

و نمودار لاس نیز به شکل زیر به دست آمد:

همانطور که دیده می‌شود، استفاده از RMSprop در تفکیک کننده سبب شده است که به یک حالت پایدار همگرا شود و در کل می‌توان گفت که همگرایی در شبکه رخ داده است. همچنین با توجه به تصویر خروجی نیز می‌توان گفت که Mode collapse نیز رخ نداده است.



شکل ۷۳ نمودار لاس شبکه WGAN-GP با تابع هزینه Adam و RMSprop و برعکس کردن لیبلها (مسئله بیشینه‌سازی)

در کل می‌توانیم بگوییم که شبکه‌های GAN به شدت نسبت به هایپرپارامترها حساس هستند و خروجی‌های زیادی باید گرفته شود تا بتوان شبکه را به یک حالت پایدار رساند.

سوال ۳ – VQ-VAE

الف)

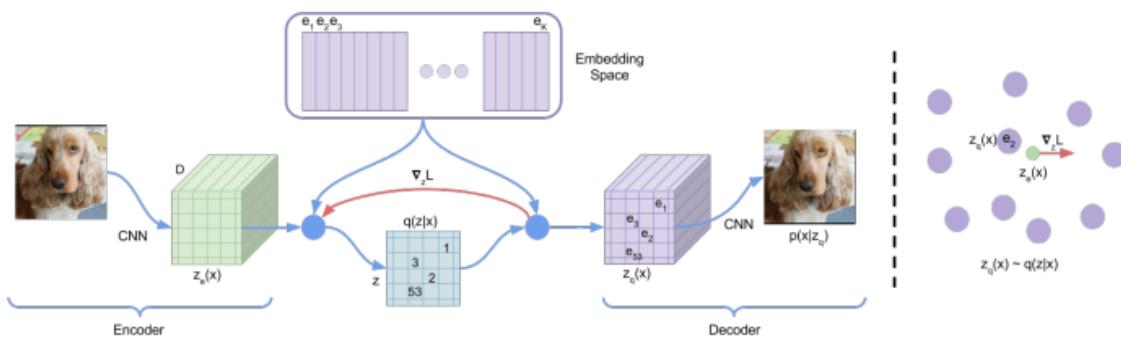
همانطور که از نام این شبکه پیداست، مهم‌ترین تفاوت آن با یک VAE معمولی، کوانتیزه کردن بردارهای امبدینگ است و درواقع به جای تخمین یک توزیع پیوسته از کدگذاری گسسته استفاده می‌شود. درواقع در VQ VAE به جای تخمین یک توزیع پیوسته بر روی داده‌ها، یک توزیع گسسته یادگیری می‌شود. تفاوت دوم و مهم این است که توزیع prior به جای اینکه یک چیز ثابت در نظر گرفته شده باشد، خود نیز یادگیری می‌شود. برای درک بهتر این موضوع در ادامه توضیحاتی ارائه خواهیم کرد.

مهم‌ترین بخش یک انکودر و دیکودر، فضای latent آن است که در آن توزیع داده‌ها به دست می‌آید و درواقع داده اصلی، یک تبدیل خطی یا غیر خطی از این فضا است. با دسترسی به این فضای latent می‌توان ویژگی‌های اصلی داده‌ها را به دست آورد که می‌تواند ابعاد کوچکتری نیز داده باشد و درواقع داده‌ها فشرده شده‌اند و داده‌های نویز نیز حذف شده است. اگر بتوان داده‌های latent را با تبدیل خطی به فضای واقعی برد، در این موارد می‌توان از الگوریتم‌های ساده‌ای مانند PCA برای فشرده کردن داده‌ها به دست آورد. اما اگر این تبدیل‌ها غیر خطی باشد، نیاز است که از الگوریتم‌های پیچیده‌تری مانند انکودرها استفاده کنیم. حال اگر بخواهیم با استفاده از توزیعات به دست آمده در لایه latent داده‌های جدیدتری تولید کنیم، لازم است که قیدی بر روی توزیع‌ها اعمال کنیم تا امکان ادغام این توزیع‌ها با یکدگیر وجود داشته باشد که در این صورت یک VAE تولید می‌شود.

در یک VAE سعی می‌شود که در فضای latent توزیع‌ها به نحوی به دست آیند که در مجاورت یکدگیر قرار گرفته باشند. در حالت ایده‌آل تمام توزیع‌ها در یک ناحیه کوچک متمرکز می‌شوند و کل فضای latent را پوشش نمی‌دهند. در بیشتر VAE‌ها فرض می‌شود که داده‌ها از یک توزیع نرمال با میانگین μ و واریانس σ^2 به دست می‌آید. همچنین با فرض اینکه داده‌های خام یک متغیر تصادفی x باشد، یک توزیع posterior به شکل $p(z|x)$ تعریف می‌شود. هدف نهایی این است که این این توزیع برای داده‌ها محاسبه شود. اما چون $p(x|z)$ یک توزیع پیوسته است، امکان اینکه بتوانیم تمام مقادیر آن را محاسبه کنیم وجود ندارد. بنابراین لازم است که یک قید بر روی این توزیع اعمال شود و در نظر گرفته می‌شود که توزیع‌ها، توزیع‌های گاوی مستقل هستند. این توزیع تخمین زده شده $p(z|x)$ نامیده می‌شود. حال هدف این است که تفاوت میان توزیع داده‌ها و توزیع تخمین‌زده شده به کمترین میزان خود برسد.

مزیت VQ VAE در این است که دیگر نیازی به KL Divergence وجود ندارد. در نتیجه مشکل collapse posterior از بین می‌رود چراکه دیگر مقدار KL به صفر نزدیک نخواهد شد. این مشکل به خصوص زمانی پیش می‌آید که زمانی که دیکودر بیش از اندازه قدرتمند باشد، لایه‌های latent در نظر گرفته نمی‌شوند [6].

نمایی از یک شبکه VQ VAE در تصویر زیر دیده می‌شود:



[6] ۷۴ ساختار شبکه VQ VAE

در بخش بعدی به نحوه عملکرد این شبکه خواهیم پرداخت.

(ب)

در یک VQ VAE با افزودن یک Codebook گستته بعد از انکودر، می‌توانیم عملکرد انکودر را گسترش دهیم. یک Codebook در واقع لیستی از بردارها است که هر بردار یک شماره مخصوص به خود را دارد و برای گستته‌سازی لایه میانی اتوانکودر به کار برده می‌شود. پس از هر بار محاسبه خروجی در انکودر، این خروجی با تمام بردارهای موجود در این لیست مقایسه می‌شود و برداری که کمترین فاصله اقلیدسی را با خروجی داشته باشد به عنوان ورودی به دیکودر داده می‌شود. در واقع بردار مطلوب به این شکل انتخاب می‌شود:

$$z_q(x) = \operatorname{argmin}_i \| z_e(x) - e_i \|_2$$

درواقع در اینجا شماره برداری که کمترین فاصله را دارد پیدا می‌شود. اما مشکلی که این فرآیند ایجاد می‌کند این است که argmin یک عملیات غیر مشتق‌پذیر است و بنابراین نمی‌توانیم به طور مستقیم از این قسمت مشتق‌گیری کنیم و در واقع در backpropagation مشتق انکودر از دست می‌رود. برای رفع این مشکل به طور مستقیم گرادیان‌های دیکودر کپی می‌شوند و در واقع فرض می‌شود که گرادیان این عملیات برابر با یک بوده است.

موردی که اینجا می‌تواند مورد سوال واقع شود این است که دیکودر چگونه تنها با تعداد بردارهای محدودی می‌تواند داده اصلی را بازیابی کند. اما نکته اینجاست که انکودر تنها یک بردار را به عنوان خروجی ایجاد نمی‌کند. درواقع یک انکودر می‌تواند یک لیست از بردارها یا یک ماتریس بردارها را تولید کند. پس از آن هر کدام از این بردارها جداگانه گسترش‌سازی می‌شوند و سپس به دیکودر داده می‌شوند.

نکته بعدی نحوه تعریفتابع هزینه است. برای یک شبکه VQVAE تابع هزینه به شکل زیر تعریف می‌شود:

$$\log(p(x | q(x))) + \| \text{sg}[z_e(x)] - e \|_2^2 + \beta \| z_e(x) - \text{sg}[e] \|_2^2$$

در اینجا sg به معنای stop gradient است و مانع از اعمال گرادیان در این معادله می‌شود. قسمت اول این تابع هزینه مربوط به تابع هزینه reconstruction است که در شبکه‌های مشابه دیگر نیز وجود دارد. قسمت دوم مربوط به codebook alignment loss است که هدف از آن، انتخاب برداری است که کمترین فاصله را با ورودی دارد. همانطور که دیده می‌شود sg بر روی خروجی انکودر قرار گرفته است چراکه هدف این لاس، بهبود codebook است. قسمت سوم نیز مربوط به commitment loss است. تنها تفاوت بین قسمت دوم و سوم، محل قرار دادن sg است. در قسمت سوم هدف حل مسئله معکوس است که در آن نیاز است که خروجی انکودر بازسازی شود.^۱

(ج)

در ابتدا ساختار شبکه پیاده‌سازی شده در keras را توضیح میدهیم. در ابتدا دیتابست mnist مورد استفاده قرار گرفته است و داده‌های آن نرمالایز شدند. پس از آن لایه‌های VQ که لایه‌های latent محسوب می‌شوند به شکل زیر تعریف شد. این تکه کد از آموزش موجود در سایت Keras الگوبرداری شده است که در ادامه ریزجزئیات آن توضیح داده خواهد شد.^۲

```
class VectorQuantizer(layers.Layer):
    def __init__(self, num_embeddings, embedding_dim, beta=0.25, **kwargs):
        super().__init__(**kwargs)
        self.embedding_dim = embedding_dim
        self.num_embeddings = num_embeddings
        self.beta = beta

        w_init = tf.random_uniform_initializer()
        self.embeddings = tf.Variable(
            initial_value=w_init(
                shape=(self.embedding_dim, self.num_embeddings), dtype="float32"
            ),
            trainable=True,
            name="embeddings_vqvae",
        )
```

شکل ۷۵ کلاس مربوط به لایه VQ

همانطور که دیده می‌شود در ابتدا وزن‌های اولیه و لایه کدینگ اولیه تولید می‌شود. از آنجایی که این کلاس یک زیر کلاس از Layer محسوب می‌شود، باید متده Call را داشته باشد که در هر ران شبکه برای دادن خروجی مورد استفاده قرار می‌گیرد. این متده در تصویر زیر دیده می‌شود.

```
def call(self, x):
    """
    transformation from inputs to outputs
    Calculate the input shape of the inputs and
    then flatten the inputs keeping `embedding_dim` intact.
    """
    input_shape = tf.shape(x)
    flattened = tf.reshape(x, [-1, self.embedding_dim])

    # Quantization.
    encoding_indices = self.get_code_indices(flattened)
    encodings = tf.one_hot(encoding_indices, self.num_embeddings)
    quantized = tf.matmul(encodings, self.embeddings, transpose_b=True)

    # Reshape the quantized values back to the original input shape
    quantized = tf.reshape(quantized, input_shape)

    commitment_loss = tf.reduce_mean((tf.stop_gradient(quantized) - x) ** 2)
    codebook_loss = tf.reduce_mean((quantized - tf.stop_gradient(x)) ** 2)
    self.add_loss(self.beta * commitment_loss + codebook_loss)

    # Straight-through estimator.
    quantized = x + tf.stop_gradient(quantized - x) # only inputs will be remained
    return quantized
```

شکل ۷۶ متده call در لایه VQ

مهم‌ترین بخش در این قسمت مربوط به خط آخر می‌شود. نکته این است که عملیات گسسته سازی عملی غیر قابل گرادیان‌گیری است و در چنین حالتی، ما عملاً نمی‌توانیم از گرادیان‌ها در بخش انکودر استفاده کنیم. به همین منظور با در نظر نگرفتن گرادیان‌های این بخش، گرادیان‌های بخش دیکودر را به طور مستقیم به بخش انکودر منتقل می‌کنیم. با استفاده از آخرین خط کد نیز همین عملیات را درواقع انجام می‌دهیم و با غیر فعال‌سازی گرادیان میان ورودی و خروجی، گرادیان ورودی تنها باقی می‌ماند.

مابقی قسمت‌های کد نیز برای پیداکردن مکان انکودینگ مورد استفاده قرار می‌گیرد و با one-hot coding code book مدنظر تشکیل می‌شود که در فضایی گسسته قرار می‌گیرند.

همچنین مقدار loss نیز برابر با مقدار loss گفته شده در مقاله در نظر گرفته می‌شود. پس از این قسمت لازم است که لایه انکودر و دیکودر تعریف شوند. در ابتدا لایه انکودر و دیکودر مخصوص دیتاست MNIST توضیح داده می‌شود و سپس به دیتاست CIFAR خواهیم پرداخت. برای دیتاست MNIST از دو لایه کانولوشنی با تعداد فیلتر ۳۲ و ۶۴ برای لایه انکودر و ۳۲ و ۶۴ برای دیکودر استفاده شده است و در لایه میانی نیز لایه VQ قرار می‌گیرد.

```

def get_encoder(latent_dim=16):
    encoder_inputs = keras.Input(shape=(28, 28, 1))
    x = layers.Conv2D(32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
    x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
    encoder_outputs = layers.Conv2D(latent_dim, 1, padding="same")(x)
    return keras.Model(encoder_inputs, encoder_outputs, name="encoder")

def get_decoder(latent_dim=16):
    latent_inputs = keras.Input(shape=get_encoder(latent_dim).output.shape[1:])
    x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(latent_inputs)
    x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
    decoder_outputs = layers.Conv2DTranspose(1, 3, padding="same")(x)
    return keras.Model(latent_inputs, decoder_outputs, name="decoder")

```

شکل ۷۷ تعریف لایه‌های انکودر و دیکورد برای دیتاست **MNIST**

ساختار انکودر و دیکورد در دو تصویر زیر دیده می‌شود.

Model: "encoder"		
Layer (type)	Output Shape	Param #
input_5 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d_6 (Conv2D)	(None, 14, 14, 32)	320
conv2d_7 (Conv2D)	(None, 7, 7, 64)	18496
conv2d_8 (Conv2D)	(None, 7, 7, 16)	1040

شکل ۷۸ ساختار انکودر برای **MNIST**

Model: "decoder"		
Layer (type)	Output Shape	Param #
input_7 (InputLayer)	[(None, 7, 7, 16)]	0
conv2d_transpose_3 (Conv2DT ranspose)	(None, 14, 14, 64)	9280
conv2d_transpose_4 (Conv2DT ranspose)	(None, 28, 28, 32)	18464
conv2d_transpose_5 (Conv2DT ranspose)	(None, 28, 28, 1)	289

شکل ۷۹ ساختار دیکورد برای **MNIST**

در نهایت نیز می‌توانیم مدل اصلی را تعریف کنیم. در اینجا مقدار ۱۶۴ امبدینگ (عمق کد بوک) و ابعاد لایه latent نیز برابر با ۱۶ برای دیتاست **MNIST** در نظر گرفته شده است. درواقع در هر ردیف ۱۶ تایی، یک ستون ۶۴ تایی وجود دارد که one hot encode شده است.

حال لازم است فرآیند یادگیری را برای این شبکه انجام دهیم. به همین منظور کلاس VQVAEModel توسعه داده شده است که زیر کلاسی از Model است.

```
class VQVAEModel(keras.models.Model):
    def __init__(self, train_variance, latent_dim, num_embeddings, **kwargs):
        super(VQVAEModel, self).__init__(**kwargs)
        self.train_variance = train_variance
        self.latent_dim = latent_dim
        self.num_embeddings = num_embeddings

        self.vqvae = get_vqvae(self.latent_dim, self.num_embeddings)

        self.total_loss_tracker = keras.metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = keras.metrics.Mean(
            name="reconstruction_loss"
        )
        self.vq_loss_tracker = keras.metrics.Mean(name="vq_loss")
```

شکل ۸۰ کلاس VQVAEModel

همانطور که دیده می‌شود در ابتدا loss های مختلف شبکه در این کلاس تعریف می‌شود. یک لاس مربوط به Reconstruction loss می‌شود که در واقع برای بهینه‌سازی انکودر و دیکودر مورد استفاده قرار می‌گیرد. اما لاس دوم مربوط به لایه‌های VQ است که به آن Codebook loss نیز گفته می‌شود. این لاس درواقع فاصله میان هر ورودی و امبدینگ‌ها را می‌سنجد تا بتواند نزدیک‌ترین امبدینگ را پیدا کند. یک گام یادگیری در متدهای train_step پیاده‌سازی شده است که در تصویر زیر نیز قرار گرفته است.

```
def train_step(self, x):
    with tf.GradientTape() as tape:
        reconstructions = self.vqvae(x)
        reconstruction_loss = (
            tf.reduce_mean((x - reconstructions) ** 2) / self.train_variance
        )
        total_loss = reconstruction_loss + sum(self.vqvae.losses)

    grads = tape.gradient(total_loss, self.vqvae.trainable_variables)
    self.optimizer.apply_gradients(zip(grads, self.vqvae.trainable_variables))

    self.total_loss_tracker.update_state(total_loss)
    self.reconstruction_loss_tracker.update_state(reconstruction_loss)
    self.vq_loss_tracker.update_state(sum(self.vqvae.losses))

    return {
        "loss": self.total_loss_tracker.result(),
        "reconstruction_loss": self.reconstruction_loss_tracker.result(),
        "vqvae_loss": self.vq_loss_tracker.result(),
    }
```

شکل ۸۱ یک گام یادگیری

همانطور که دیده می‌شود در اینجا از تخمینی به عنوانتابع هزینه Reconstruction استفاده شده است که در بخش بعدی این تابع تغییر پیدا خواهد کرد. لازم به ذکر است که در اینجا فرض شده است که داده‌ها از توزیعی نرمال به دست آمده‌اند.

$$\log P(x | z) = -\frac{1}{2} [\log(|\Sigma|) + k \log(2\pi) + (\mathbf{x} - \mu)^T (\mathbf{x} - \mu)]$$

در نتیجه دو بخش اول ثابت هستند و تنها بخش اخر باقی می‌ماند که به شکل بالا در کد تخمین زده شده است که در واقع یک MSEloss ساده محاسبه می‌شود.

بخش دوم loss مربوط به لایه VQ است که خود از دو بخش تشکیل شده است. بخش اول مربوط به بخش دوم commitment_loss است که به شکل زیر تعریف می‌شود:

$$\beta \|z_e(x) - \text{sg}[e]\|_2^2$$

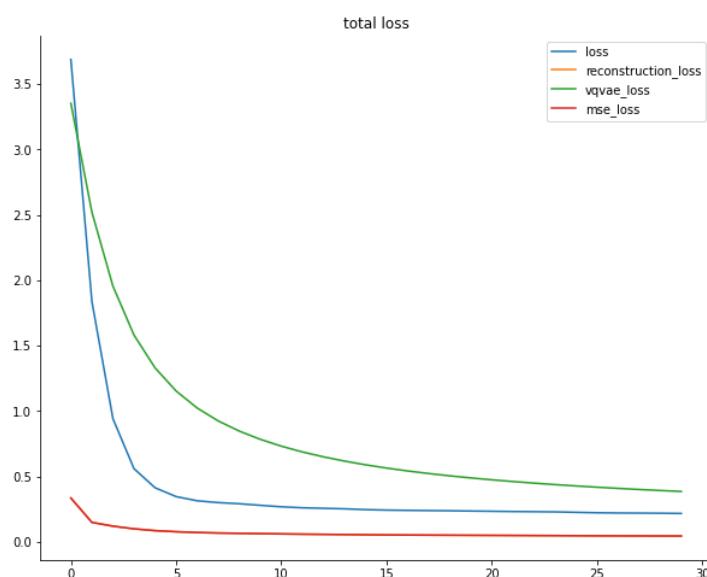
و بخش دوم مربوط به codebook loss است که به شکل زیر تعریف می‌شود.

$$\|\text{sg}[z_e(x)] - e\|_2^2$$

که این دو تابع هزینه در خود لایه VQ تعریف شده‌اند.

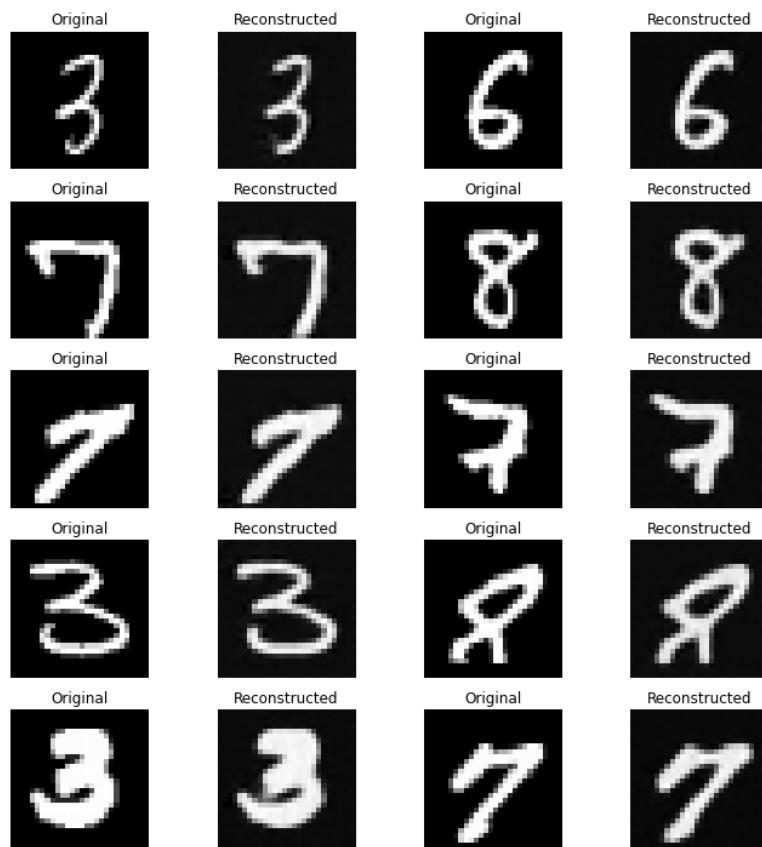
در مقاله گفته شده است که تغییر مقدار بتا از ۰.۱ تا ۰.۲ تغییر محسوسی را به همراه نداشته است و توصیه شده است که از مقدار ۰.۲۵ استفاده شود. ما نیز از همین مقدار در این مسئله استفاده کردیم.

با آموزش شبکه بر روی دیتاست MNIST نمودار loss زیر به دست آمد.



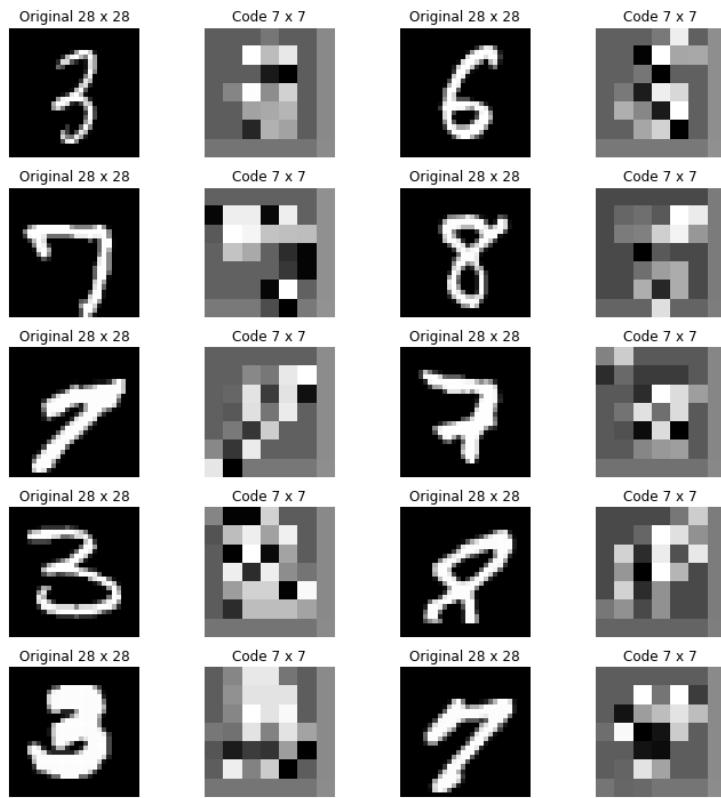
شکل ۸۲ مقدار تابع هزینه در ۳۰ اپاک برای دیتاست MNIST

همچنین ورودی شبکه و خروجی‌های دیکودر در تصویر زیر قرار گرفته است:



شکل ۸۳ ورودی و خروجی شبکه **VQ VAE** برای دیتاست **MNIST**

همچنین برای درک بهتر خروجی لایه‌های امبدینگ، خروجی آن‌ها در تصویر زیر رسم شده است:



شکل ۸۴ لایه‌های کدگذاری شده برای دیتاست MNIST

در این تصاویر نیز دیده می‌شود که برخی ویژگی‌های متمایز هر تصویر در این کدگذاری‌ها مشخص شده است.

حال همین شبکه را با اعمال برخی تغییرات بر روی شبکه بر روی دیتاست CIFAR10 نیز پیاده‌سازی می‌کنیم. تغییر اولی که ایجاد شده است، تغییرات ابعاد لایه latent و تعداد امبدینگ‌ها است. در اینجا چون با تصاویری رنگی و ابعاد بزرگتری روبرو هستیم، ابعاد لایه latent برابر با ۶۴ و ۵۱۲ امبدینگ برای این لایه (عمق) انتخاب شد.

تغییر دوم، افزودن لایه‌های کانولوشنی به انکودر و دیکودر است تا ویژگی‌های تصاویر به نحو بهتری استخراج شود. به همین منظور یک لایه کانولوشنی با stride ۱۲۸ فیلتر با ۲ و یک لایه کانولوشنی با ۲۵۶ لایه و stride ۱ برابر با ۱ به انکودر اضافه شد. در دیکودر نیز یک لایه به عمق ۲۵۶ و stride ۱ و یک لایه به عمق ۱۲۸ با stride ۲ به دیکودر افزوده شد. همچنین چون با تصاویری سه کanalه روبرو هستیم، ابعاد ورودی نیز تغییر پیدا کرد تا بتوان از هر سه لایه تصویر استفاده کرد. ساختار انکودر در تصویر زیر قرار گرفته است:

Model: "encoder"		
Layer (type)	Output Shape	Param #
input_5 (InputLayer)	[(None, 32, 32, 3)]	0
conv2d_10 (Conv2D)	(None, 32, 32, 32)	896
conv2d_11 (Conv2D)	(None, 16, 16, 64)	18496
conv2d_12 (Conv2D)	(None, 8, 8, 128)	73856
conv2d_13 (Conv2D)	(None, 8, 8, 256)	295168
conv2d_14 (Conv2D)	(None, 8, 8, 64)	16448

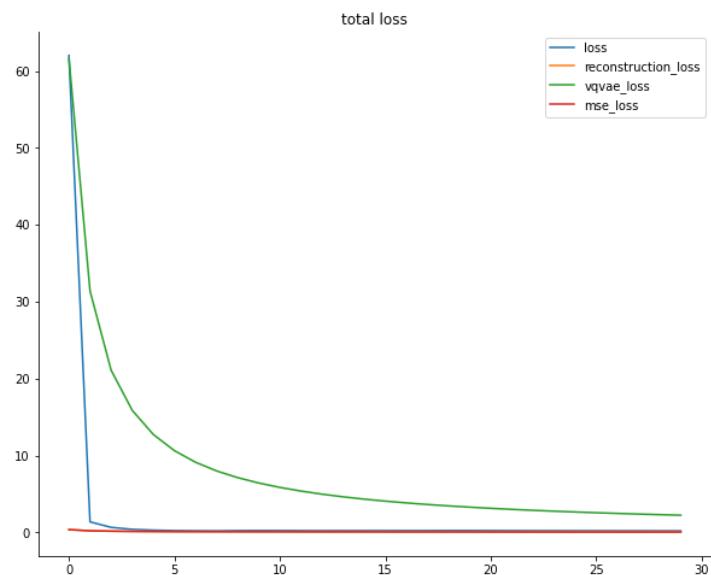
شکل ۸۵ ساختار انکودر برای دیتاست **CIFAR10**

و ساختار دیکودر نیز در تصویر زیر قرار گرفته است:

Model: "decoder"		
Layer (type)	Output Shape	Param #
input_7 (InputLayer)	[(None, 8, 8, 64)]	0
conv2d_transpose_4 (Conv2DT transpose)	(None, 8, 8, 256)	147712
conv2d_transpose_5 (Conv2DT transpose)	(None, 16, 16, 128)	295040
conv2d_transpose_6 (Conv2DT transpose)	(None, 32, 32, 64)	73792
conv2d_transpose_7 (Conv2DT transpose)	(None, 32, 32, 3)	1731

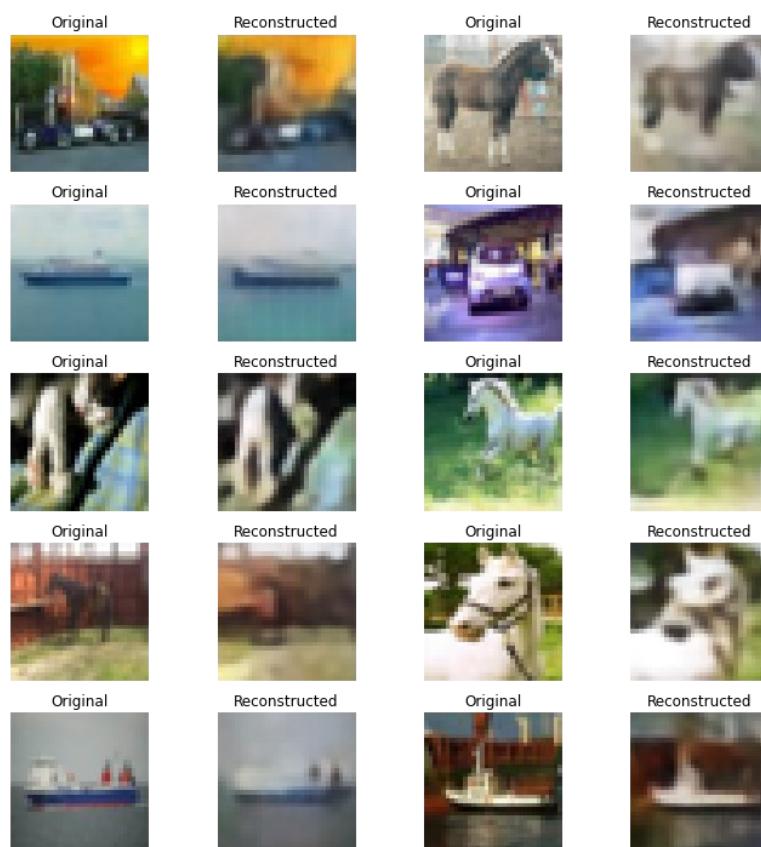
شکل ۸۶ ساختار دیکودر برای دیتاست **CIFAR10**

با در نظر گرفتن این معماری شبکه، نمودار لاس به شکل زیر به دست آمده است:



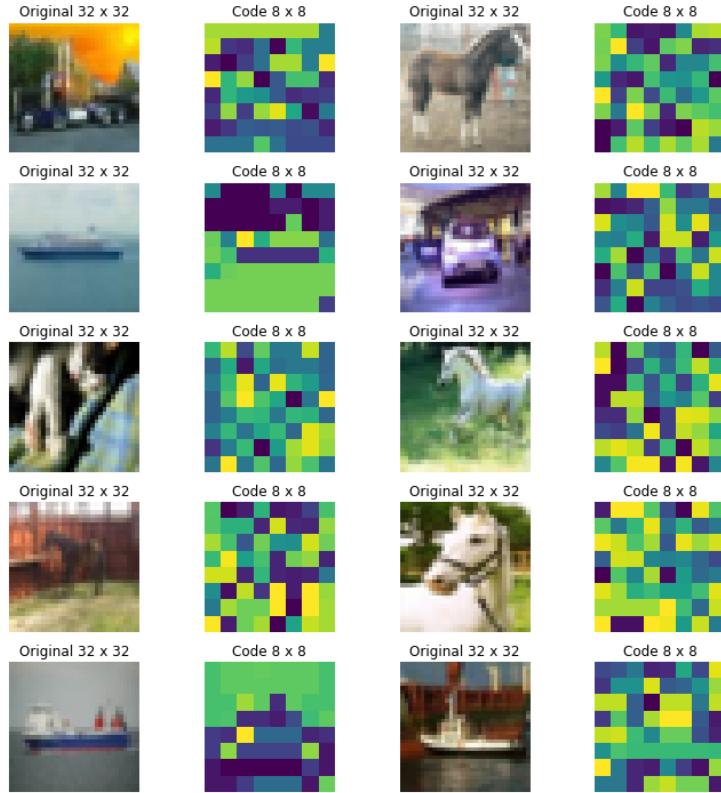
شکل ۸۷ مقدار توابع هزینه در ۳۰ ایپاک برای دیتاست CIFAR10

و خروجی‌های دیکودر و ورودی شبکه نیز در تصویر زیر رسم شده است:



شکل ۸۸ ورودی و خروجی شبکه VQ VAE برای دیتاست CIFAR10

همچنین برای درک بهتر خروجی لایه‌های امبدینگ، خروجی آن‌ها در تصویر زیر رسم شده است:



شکل ۸۹ لایه‌های کدگذاری شده برای دیتاست CIFAR10

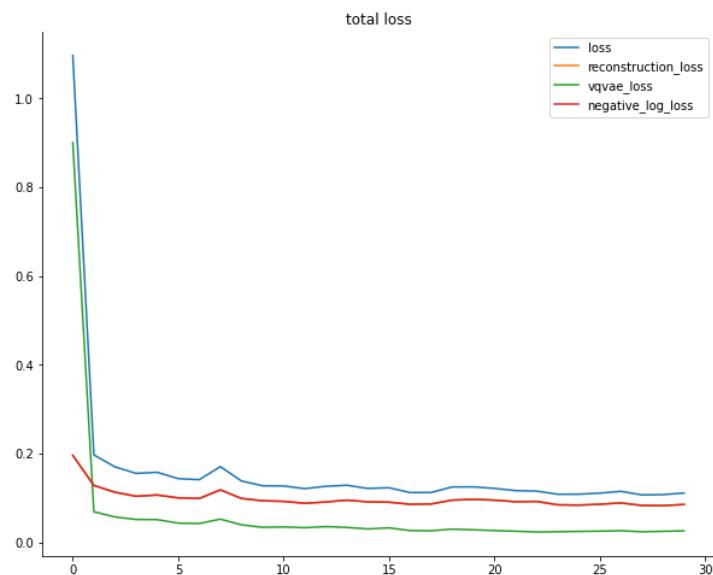
(۵)

در بخش قبلی شبکه با تابع هزینه‌ای برابر با میانگین توان دوم اختلاف ورودی و خروجی آموزش داده شد. پس در این قسمت تنها لازم است که از NLLloss استفاده شود. از آنجا که در keras تابعی دقیقا مشابه این تابع وجود ندارد و در CrossEntropy فرض می‌شود که خروجی‌ها از یک لایه Sigmoid عبور کرده‌اند، لذا لازم است که Negative log loss را به صورت دستی در کد پیاده‌سازی کنیم. این تابع هزینه به شکل زیر تعریف شده است:

```
negative_log_loss = backend.sum(backend.binary_crossentropy(x, reconstructions), axis=-1)
```

شکل ۹۰ تابع هزینه Negative log loss

مینیمم کردن این تابع هزینه مشابه با ماکزیمم کردن likelihood است. با در نظر گرفتن این تابع هزینه، خروجی شبکه بر روی دیتاست MNIST به شکل زیر به دست آمد:



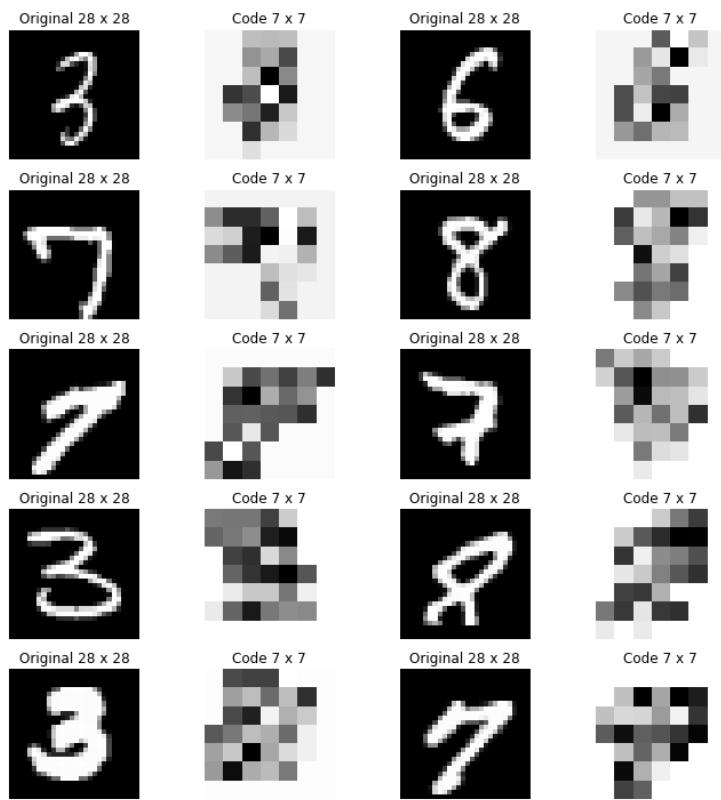
شکل ۹۱ مقدار توابع هزینه در ۳۰ ایپاک برای دیتاست **MNIST** با تابع هزینه **NLLoss**

خروجی‌های دیکودر نیز به شکل زیر به دست آمده‌اند:



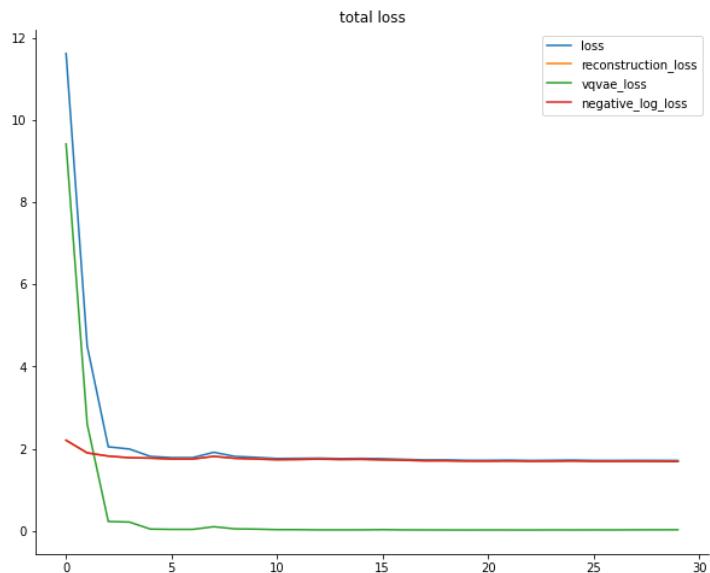
شکل ۹۲ ورودی و خروجی شبکه **VQ VAE** برای دیتاست **MNIST** با تابع هزینه **NLLoss**

و خروجی‌های لایه **VQ** نیز به شکل زیر به دست آمده است:



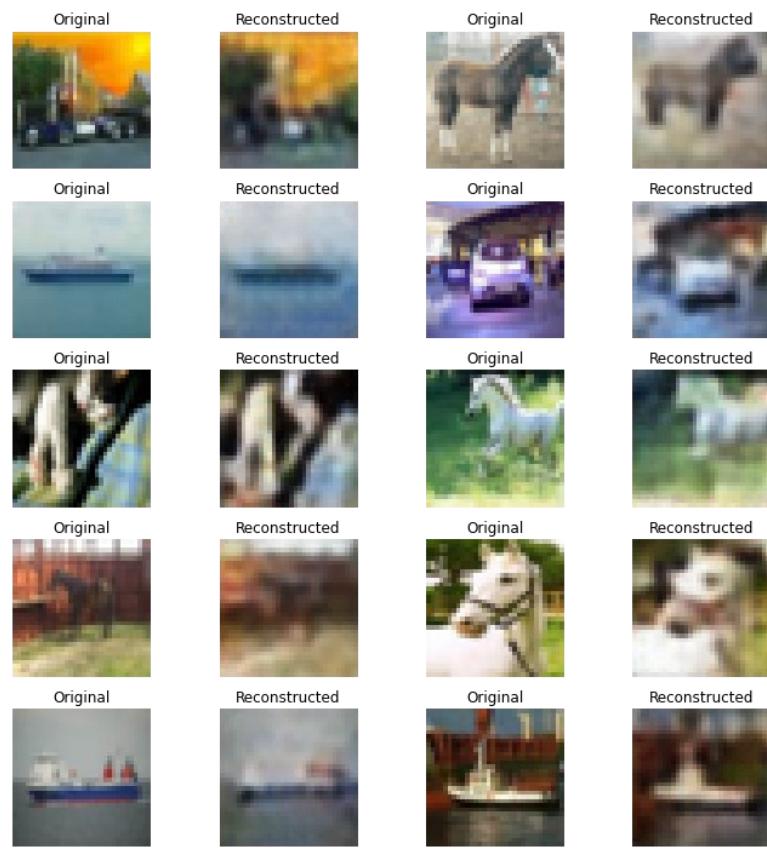
شکل ۹۳ لایه‌های کدگذاری شده برای دیتاست **MNIST** با تابع هزینه **NLLloss**

همچنین برای دیتاست **CIFAR10** نمودار لاس به شکل زیر به دست آمده است:



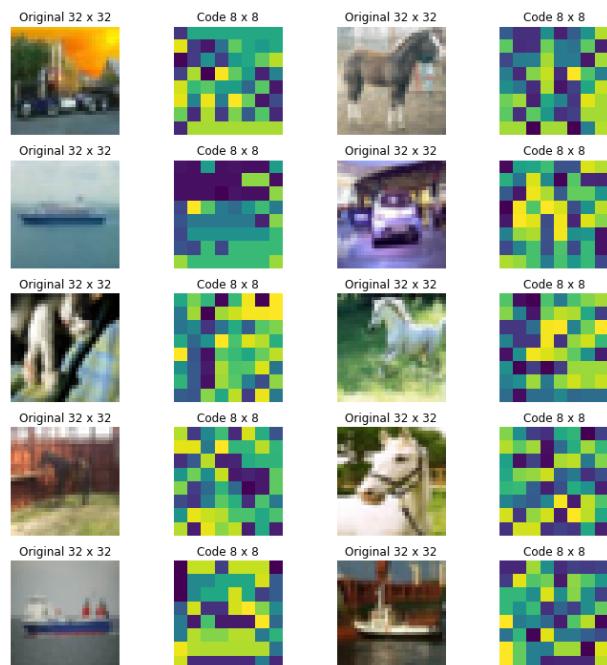
شکل ۹۴ مقدار توابع هزینه در ۳۰ ایپاک برای دیتاست **CIFAR10** با تابع هزینه **NLLloss**

خروجی لایه دیکودر نیز به شکل زیر به دست آمده است:



شکل ۹۵ ورودی و خروجی شبکه VQ VAE با تابع هزینه NLLloss برای دیتاست CIFAR10

و لایه‌های کدگذاری شده نیز به شکل زیر به دست آمدند:



شکل ۹۶ لایه‌های کدگذاری شده برای دیتاست CIFAR10 با تابع هزینه NLLloss

همانطور که دیده می‌شود بر روی هر دو دیتاست، خروجی‌ها شباهت زیادی به یکدیگیر دارند. با این حال مهم‌ترین تفاوت، در لایه VQ و امبدینگ‌ها دیده می‌شود. به نظر می‌رسد که با استفاده از NLLloss بهینه‌سازی بهتری بر روی VQ صورت گرفته است. با این حال تفاوت چندان چشم‌گیر نیست و هردو شبکه به خوبی توانسته‌اند به خروجی‌های مطلوبی دست پیدا کنند. به طور کلی نیز گفته می‌شود که MSEloss برای بهینه‌سازی کمی دشوار است و در برخی موارد می‌تواند منجر به عدم همگرایی شود.

همچنین با دقت در لایه خروجی دو شبکه، می‌توانیم متوجه شویم که با استفاده از NLLloss، تصاویر مقداری محotor هستند و به نظر می‌رسد که مقداری از اطلاعات مهم دیتاست از دست رفته است. این موضوع به خصوص در دیتاست MNIST بیشتر قابل مشاهده است و تصاویر به نحوی هستند که پس از بازیابی، مقداری نویز به آن‌ها افزوده شده است.

مراجع

- [1] M. Arjovsky and L. Bottou, “Towards principled methods for training generative adversarial networks,” *5th Int. Conf. Learn. Represent. ICLR 2017 - Conf. Track Proc.*, pp. 1–17, 2017.
- [2] K. Roth, A. Lucchi, S. Nowozin, and T. Hofmann, “Stabilizing training of generative adversarial networks through regularization,” *Adv. Neural Inf. Process. Syst.*, vol. 2017-Decem, no. 2, pp. 2019–2029, 2017.
- [3] M. Arjovsky, S. Chintala, and L. Bottou, “Wasserstein GAN,” 2017, [Online]. Available: <http://arxiv.org/abs/1701.07875>.
- [4] M. Y. Liu and O. Tuzel, “Coupled generative adversarial networks,” *Adv. Neural Inf. Process. Syst.*, pp. 469–477, 2016.
- [5] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, “Improved training of wasserstein GANs,” *Adv. Neural Inf. Process. Syst.*, vol. 2017-Decem, pp. 5768–5778, 2017.
- [6] A. Van Den Oord, O. Vinyals, and K. Kavukcuoglu, “Neural discrete representation learning,” *Adv. Neural Inf. Process. Syst.*, vol. 2017-Decem, no. Nips, pp. 6307–6316, 2017.