به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

شبکه های عصبی و یادگیری عمیق

مینی‌پروژه سری سوم

نام و نام خانوادگی	آرمان اکبری – شایان واصف
شماره دانشجویی	810197456 - 810197603
تاریخ ارسال گزارش	بهمن‌ماه 1400

فهرست گزارش سوالات

سوال اول – AC-GAN

در این بخش به بررسی سازوکار شبکه AC-GAN که قابلیت تولید تصاویر با در نظر گرفتن برچسب کلاس آنها را دارد می‌پردازیم.

الف)

شبکه AC-GAN افزوده‌ای بر شبکه‌های GAN است که مختصر شده Auxiliary Classifier GAN می‌باشد. تفاوت اصلی این شبکه با GAN ساده در این است که هر نمونه برای تولید[1] هر نمونه علاوه بر استفاده از یک نویز z و از یک برچسب[2] کلاس c متناظر با نویز نیز استفاده می‌کند. Generator با استفاده از این کلاس و نویز ورودی خروجی‌های $X_{fake} = G(c, z)$ را تولید می‌کند.

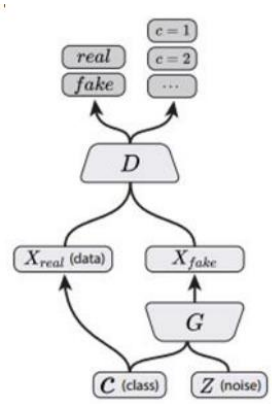
در ادامه Discriminator به هر یک از نویز و کلاس یک توزیع احتمالاتی نسبت می‌دهد:

$$P(S|X) = D(X),\ P(C|X) = D(X).$$

تابع هزینه شامل دو بخش می‌شود، لگاریتم احتمال درست تشخیص دادن نمونه واقعی که با L_s نشان داده می‌شود و لگاریتم احتمال درست تشخیص دادن کلاس‌های صحیح که با L_C نمایش داده می‌شود.

$$L_S = E[logP(S = real|X_{real})] + E\big[logP\big(S = fake|X_{fake}\big)\big]$$
$$L_C = E[logP(C = c|X_{real})] + E\big[logP\big(C = c|X_{fake}\big)\big]$$

Discriminator به دنبال حداکثر کردن $L_s + L_c$ است در حالی که $Generator$ به دنبال حداکثر کردن $L_C - L_S$ است. این شبکه دارای ساختار کلی زیر می‌باشد:



AC-GAN
(Present Work)

---

[1] Generate
[2] Label

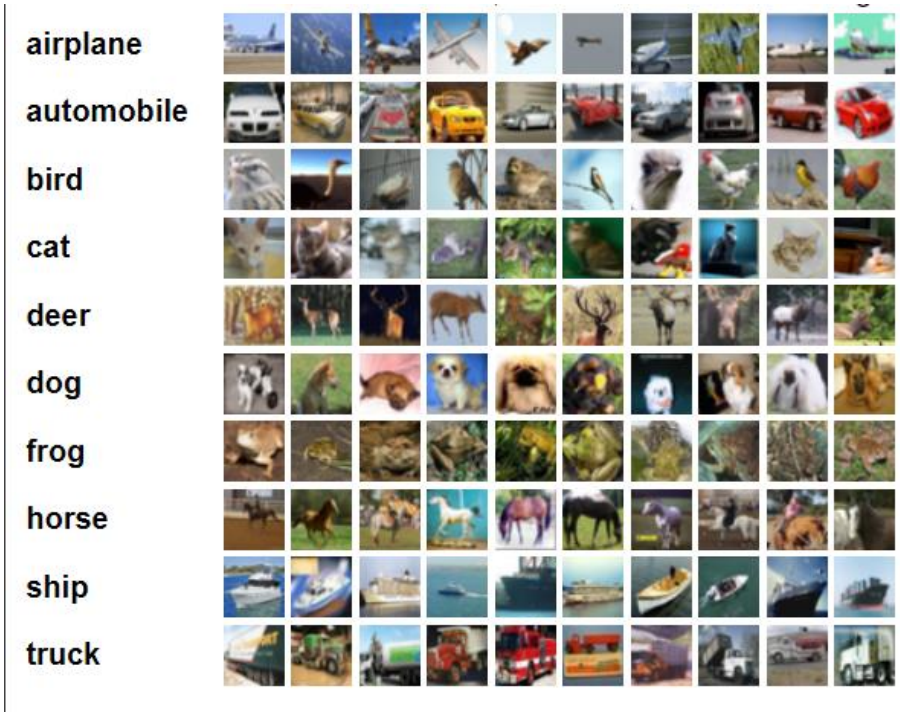
شکل 1-1. نمای ساختار کلی شبکه AC-GAN

اگر بخواهیم مقداری در ساختمان هر یک از Discriminator و Generator دقیق‌تر شویم، می‌بینیم که Generator متشکل از تعدادی شبکه دی‌کانولوشنی[1] هستند که نویز z و کلاس c را به یک تصویر تبدیل می‌کند. از طرفی Discriminator یک شبکه عصبی عمیق کانولوشنی با تابع فعالساز Leaky ReLU می‌باشد که با ورودی گرفتن تصویر اصلی و تصویر تقلبی تولید شده، علاوه بر نشان دادن اصل یا تقلبی بودن تصویر، کلاس مربوط به هر یک را نیز گزارش می‌دهد.

ب)

در این بخش به دنبال پیاده‌سازی شبکه AC-GAN بر روی دیتاست Cifar-10 هستیم. در مرحله اول نیاز داریم تا دیتاست مورد استفاده را بررسی کنیم:

دیتاست Cifar10 شامل 60000 عکس رنگی 32 در 32 در 10 کلاس می‌باشد که هر کلاس شامل 6000 عکس است. کلاس‌های تشکیل دهنده Cifar10 به صورت زیر تعریف می‌شوند:



شکل 1-2. 10 کلاس تشکیل دهنده دیتاست Cifar10

[1] Deconvolution

اکنون باید لایه‌های شبکه‌های Discriminator و Generator را طراحی کنیم. برای اینکار از مقادیر پیشنهاده شده برای دیتاست Cifar10 استفاده می‌کنیم که به شرح زیر هستند:

Operation	Kernel	Strides	Feature maps	BN?	Dropout	Nonlinearity
$G_x(z) - 110 \times 1 \times 1$ input						
Linear	N/A	N/A	384	×	0.0	ReLU
Transposed Convolution	5×5	2×2	192	√	0.0	ReLU
Transposed Convolution	5×5	2×2	96	√	0.0	ReLU
Transposed Convolution	5×5	2×2	3	×	0.0	Tanh
$D(x) - 32 \times 32 \times 3$ input						
Convolution	3×3	2×2	16	×	0.5	Leaky ReLU
Convolution	3×3	1×1	32	√	0.5	Leaky ReLU
Convolution	3×3	2×2	64	√	0.5	Leaky ReLU
Convolution	3×3	1×1	128	√	0.5	Leaky ReLU
Convolution	3×3	2×2	256	√	0.5	Leaky ReLU
Convolution	3×3	1×1	512	√	0.5	Leaky ReLU
Linear	N/A	N/A	11	×	0.0	Soft-Sigmoid
Generator Optimizer	Adam ($\alpha = [0.0001, 0.0002, 0.0003]$, $\beta_1 = 0.5$, $\beta_2 = 0.999$)					
Discriminator Optimizer	Adam ($\alpha = [0.0001, 0.0002, 0.0003]$, $\beta_1 = 0.5$, $\beta_2 = 0.999$)					
Batch size	100					
Iterations	50000					
Leaky ReLU slope	0.2					
Activation noise standard deviation	$[0, 0.1, 0.2]$					
Weight, bias initialization	Isotropic gaussian ($\mu = 0$, $\sigma = 0.02$), Constant(0)					

شکل 1-3. معماری شبکه AC-GAN برای دیتاست Cifar 10

همانطور که می‌بینیم، Generator از تعدادی لایه دی‌کانولوشنی با اندازه کرنل 5 در 5 و stride 2 در 2 تشکیل شده است. همچنین برای لایه آخر از تابع فعالساز تانژانت هایپربولیک و برای باقی لایه‌ها از تابع ReLU استفاده شده است.

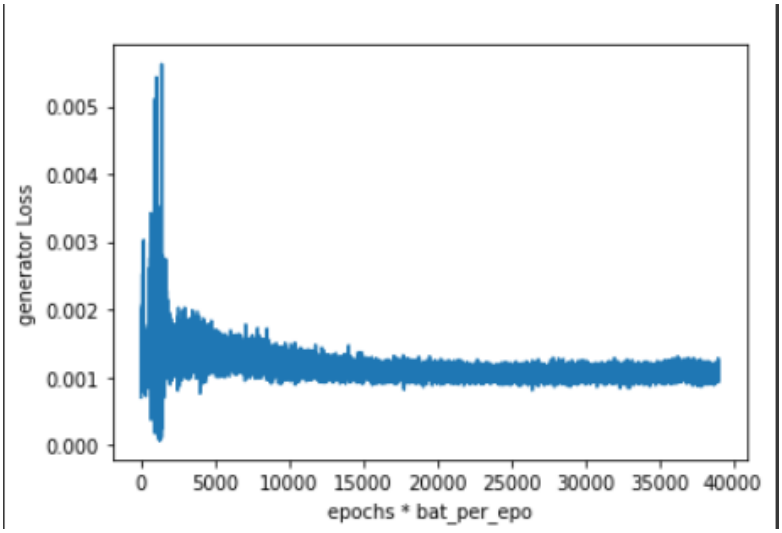
در Discriminator نیز از تعدادی لایه کانولوشنی استفاده شده است که جزییات دقیق هر یک از آنها در شکل 1-3 آمده است. همچنین پارامترهای دیگری مانند ضریب تضعیف Leaky ReLU، هایپرپارامترهای بهینه‌ساز و .. هم آورده شده است.

حال مدل خود را با کمک کتابخانه Keras در محیط پایتون پیاده سازی کرده و نتایج را بررسی می‌کنیم:

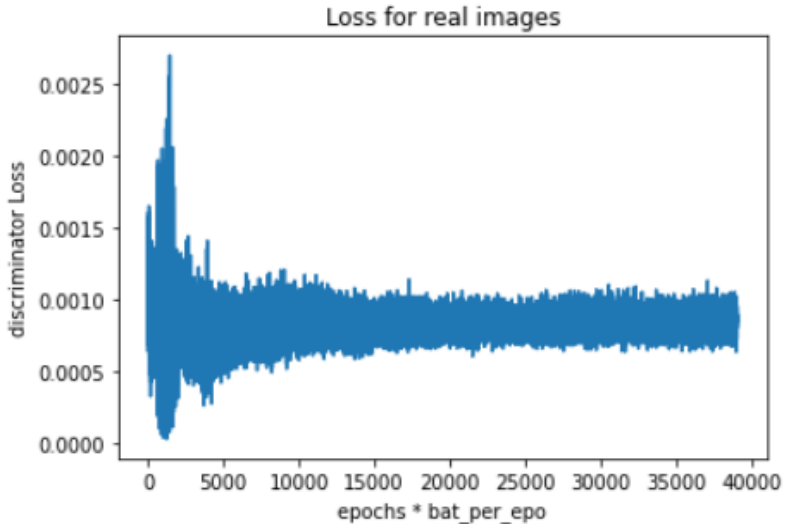
شکل 1-4. تعدادی شکل ساخته شده توسط شبکه AC-GAN برای کلاس‌های ورودی رندم

به صورت کلی دیده می‌شود که با بهم ریختگی‌هایی که انتظارش میرفت، تصاویری متناسب با ورودی‌های واقعی تولید شده است که شاید خیلی شبیه نباشند اما کلیت شکل را منتقل می‌کنند.

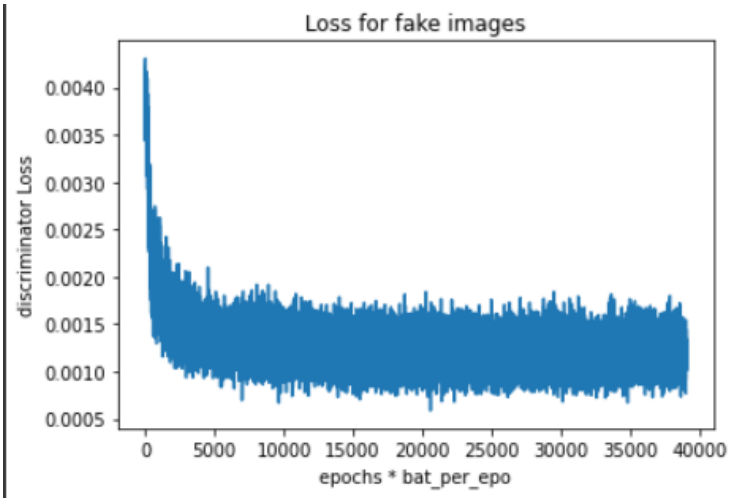
اکنون به بررسی نمودارهای خطا برای مدل ACGAN خود می‌پردازیم:



شکل 1-5. نمودار تغییرات خطا Generator شبکه در 50 ایپاک

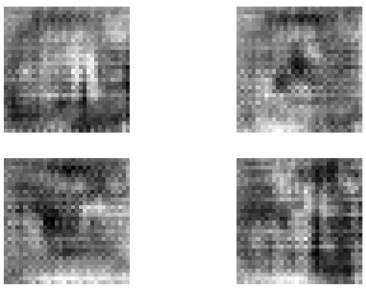


شکل 1-6. نمودار تغییرات خطا Discriminator شبکه در 50 ایپاک برای داده‌ای واقعی

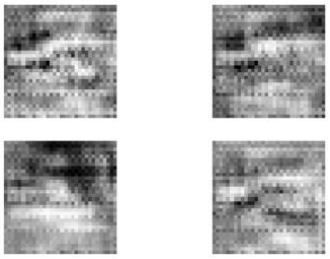


شکل 1-7. نمودار تغییرات خطا Discriminator شبکه در 50 ایپاک برای داده‌ای تقلبی

همچنین می‌توانیم خروجی شبکه را برای تعدادی از نویزهای رندوم و کلاس‌های انتخابی مشاهده کنیم:



شکل 1-8. خروجی شبکه به ازای شماره کلاس 5



شکل 1-9. خروجی شبکه به ازای شماره کلاس 1

ج) متاسفانه بدلیل کمبود وقت امکان آموزش مجدد یک ساعته برای شبکه با بهینه‌ساز RMSprop وجود نداشت.

سوال دوم- DC-GAN

الف)

در ابتدا پارامتر های مساله را تعیین می‌کنیم :

```
# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 100

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1
```

شکل 2-1 : تعیین hyperparametr های مساله

حال با داشتن directory پوشه ایجاد شده حاوی عکس ها ، به کمک Image folder ، اصلاحات مناسب را روی دادگان انجام می‌دهیم.

```
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))
```

شکل 2-2 : تعیین hyperparametr های مساله

در ادامه نمونه ای از عکس های موجود در مجموعه اصلی را می‌آوریم :

شکل 2-3 : نمونه ای از sticker های موجود

حال کلاس Generator را تعریف می‌کنیم . معماری آن به صورت زیر می‌باشد :

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

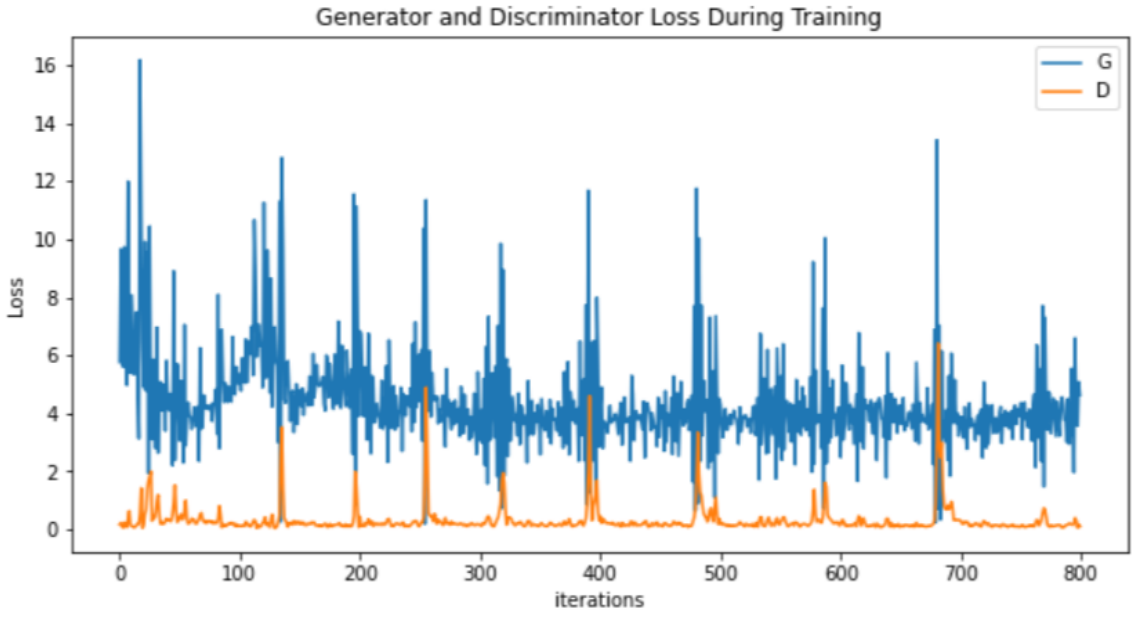
شکل 2-4 : معماری بخش Generator

به همین ترتیب ، کلاس discriminator را تعریف می‌کنیم :

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

شکل 2-4 : معماری بخش Discriminator

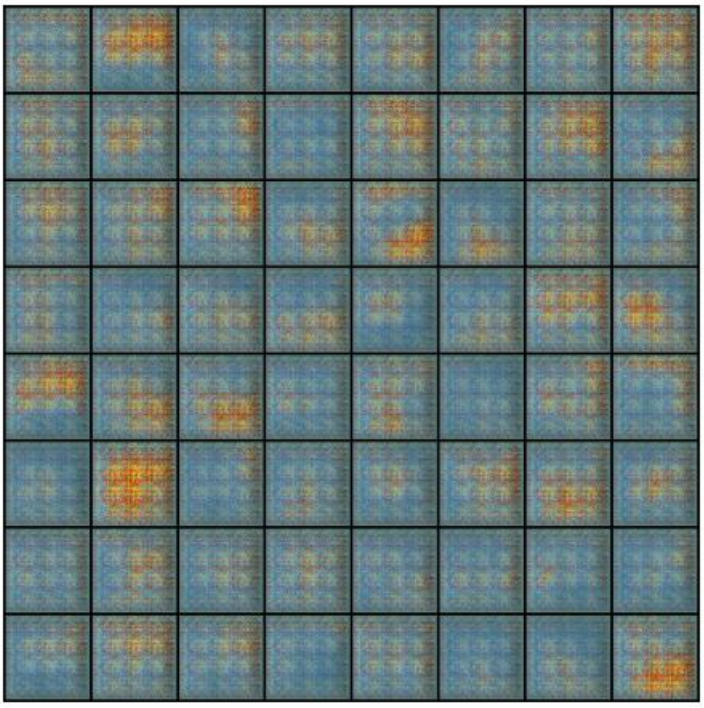
در نهایت شبکه را آموزش می‌دهیم . باید توجه شود که در ابتدا قسمت discriminator آموزش داده می‌شود و سپس قسمت Generator آموزش داده خواهد شد. بعد از 100 iteration ، نمودار discriminator و generator به صورت زیر خواهد بود :



شکل 2-5 : نحوه تغییرات خطای discriminator و Generator

طبق شکل بالا ، شاهد یک رقابت بین بخش discriminator و generator هستیم . در واقع در هر iteration این دو سعی می‌کنند عملکرد خود را بهبود یابند .

در زیر نحوه تغییرات عکس ها که از یک نویز شروع می‌شوند ، در میانه راه و در آخر نشان داده شده است :



شکل 2-6 : خروجی sticker های تولید شده در طی فرآیند آموزش

در نهایت مقایسه ای بین عکس های واقعی و عکس های fake تولیدی خواهیم داشت :

شکل 2-7 : مقایسه بین sticker های واقعی و مصنوعی تولیدی

ب)

برای پایدار سازی شبکه GAN ، چندین راهکار وجود دارد :

- به جای لایه های pooling از stride در لایه های کانولوشنی استفاده شود تا عمل down sampling را برای ما در discriminator انجام دهد (متقابلا همین استدلال در لایه های دیکانولوشنی برای generator صادق می‌باشد)
- از لایه fully connected در پایان لایه کانولوشنی استفاده نشود .
- در میان لایه ها از Batch Normalization استفاده شود .
- از activation function هایی مثل Tanh, ReLu,Leaky ReLu استفاده شود .
- از بهینه ساز Adam استفاده شود .
- روش اصلی استفاده از Gradient Penalty می‌باشد .

برای روش Gradient Penalty به تعریف زیر نیاز داریم :

A differentiable function f is 1-Lipschitz if and only if it has gradients with norm at most 1 everywhere.

f^* has gradient norm 1 almost everywhere under $\mathbb{P}_r$ and $\mathbb{P}_g$.

شکل 2-8 : برسی یک تعریف

و تابع خطا جدید به صورت زیر تعریف می‌شود :

So instead of applying clipping, WGAN-GP penalizes the model if the gradient norm moves away from its target norm value 1.

$$L = \underbrace{\mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}}\sim\mathbb{P}_g}[D(\tilde{\boldsymbol{x}})] - \mathop{\mathbb{E}}_{\boldsymbol{x}\sim\mathbb{P}_r}[D(\boldsymbol{x})]}_{\text{Original critic loss}} + \underbrace{\lambda \mathop{\mathbb{E}}_{\hat{\boldsymbol{x}}\sim\mathbb{P}_{\hat{\boldsymbol{x}}}}\left[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1)^2\right]}_{\text{Our gradient penalty}}.$$

where $\hat{\boldsymbol{x}}$ sampled from $\tilde{\boldsymbol{x}}$ and $\boldsymbol{x}$ with t uniformly sampled between 0 and 1

$$\hat{\boldsymbol{x}} = t\tilde{\boldsymbol{x}} + (1-t)\boldsymbol{x} \text{ with } 0 \le t \le 1$$

شکل 2-9 : ایجاد تابع خطا جدید

بنابراین کافی است که loss مربوط به Gradient Penalty را به loss قبلی تعریف شده که برابر جمع loss دو قسمت generator و discriminator می‌باشد را جمع کنیم .