



# I'm Something of a Painter Myself

Use GANs to create art - will you be the next Monet?

# About competition

Computer vision has advanced tremendously in recent years and GANs are now capable of mimicking objects in a very convincing way. But creating museum-worthy masterpieces is thought of to be, well, more art than science. So can (data) science, in the form of GANs, trick classifiers into believing you've created a true Monet? That's the challenge you'll take on!

Your task is to generate 7,000 to 10,000 Monet-style images.



A GAN consists of at least two neural networks: a generator model and a discriminator model. The generator is a neural network that creates the images. For our competition, you should generate images in the style of Monet. This generator is trained using a discriminator.

The two models will work against each other, with the generator trying to trick the discriminator, and the discriminator trying to accurately classify the real vs. generated images.

You are going to generate 7,000-10,000 Monet-style images that are in jpg format. Their sizes should be 256x256x3 (RGB). Then you need to zip those images and your output from your Kernel should only have ONE output file named images.zip.

Metric - MiFID

The smaller MiFID is, the better your generated images are.

# FID

In FID, we use the Inception network to extract features from an intermediate layer. Then we model the data distribution for these features using a multivariate Gaussian distribution with mean  $\mu$  and covariance  $\Sigma$ . The FID between the real images  $r$  and generated images  $g$  is computed as:

$$\text{FID} = \|\mu_r - \mu_g\|^2 + \text{Tr}(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{1/2})$$

where  $\text{Tr}$  sums up all the diagonal elements. FID is calculated by computing the Fréchet distance between two Gaussians fitted to feature representations of the Inception network.

# MIFID

In addition to FID, Kaggle takes training sample memorization into account.

The memorization distance is defined as the minimum cosine distance of all training samples in the feature space, averaged across all user generated image samples. This distance is thresholded, and it's assigned to 1.0 if the distance exceeds a pre-defined epsilon.

where  $f_g$  and  $f_r$  represent the generated/real images in feature space (defined in pre-trained networks); and  $f_{gi}$  and  $f_{rj}$  represent the  $i$  and  $j$  vectors of  $f_g$  and  $f_r$ , respectively.

$$d_{ij} = 1 - \cos(f_{gi}, f_{rj}) = 1 - \frac{f_{gi} \cdot f_{rj}}{|f_{gi}| |f_{rj}|}$$

$$d = \frac{1}{N} \sum_i \min_j d_{ij}$$

defines the minimum distance of a certain generated image ( $i$ ) across all real images ( $j$ ), then averaged across all the generated images.

threshold of the weight only applies when the ( ) is below a certain empirically determined threshold.

$$d_{thr} = \begin{cases} d, & \text{if } d < \epsilon \\ 1, & \text{otherwise} \end{cases}$$

$$MiFID = FID \cdot \frac{1}{d_{thr}}$$

Step 2000: Generator loss: 4.834059171676642, discriminator loss: 0.017098796288482824  
Mean loss: 2.4255789839825623



## Basic GAN for Monet competition

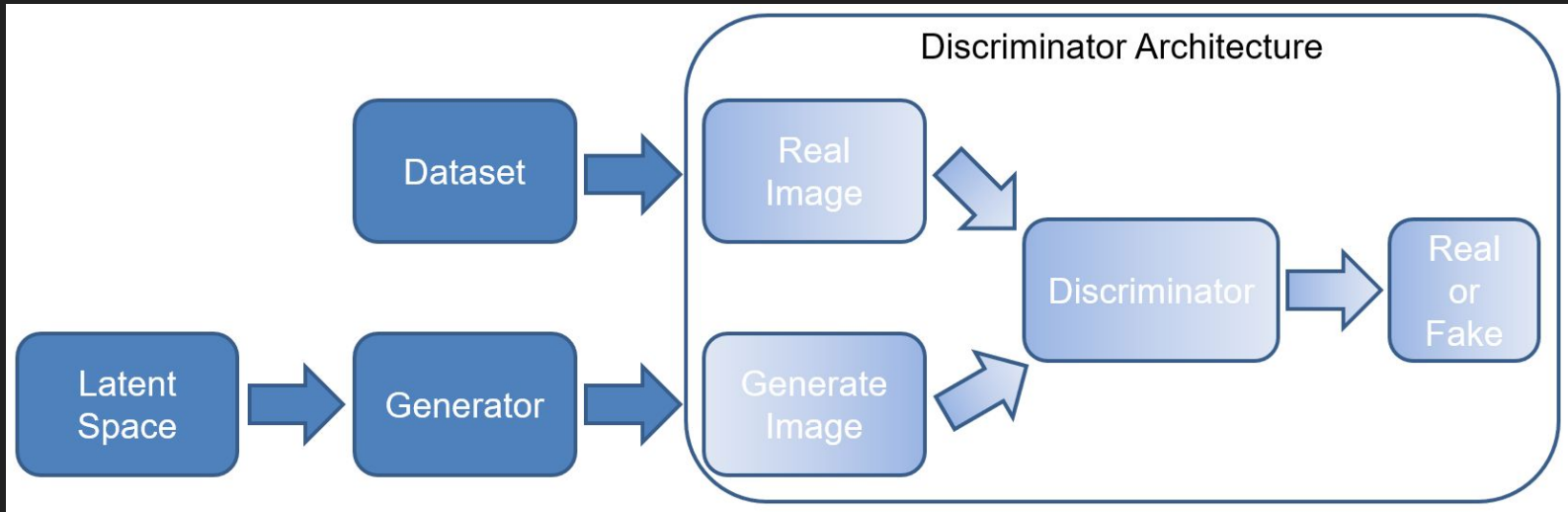
Basic non-cycle GAN on torch (version 3/3)

a day ago by [Roman Derunets](#)

Notebook Basic GAN for Monet competition | Version 3

Succeeded

276.19610



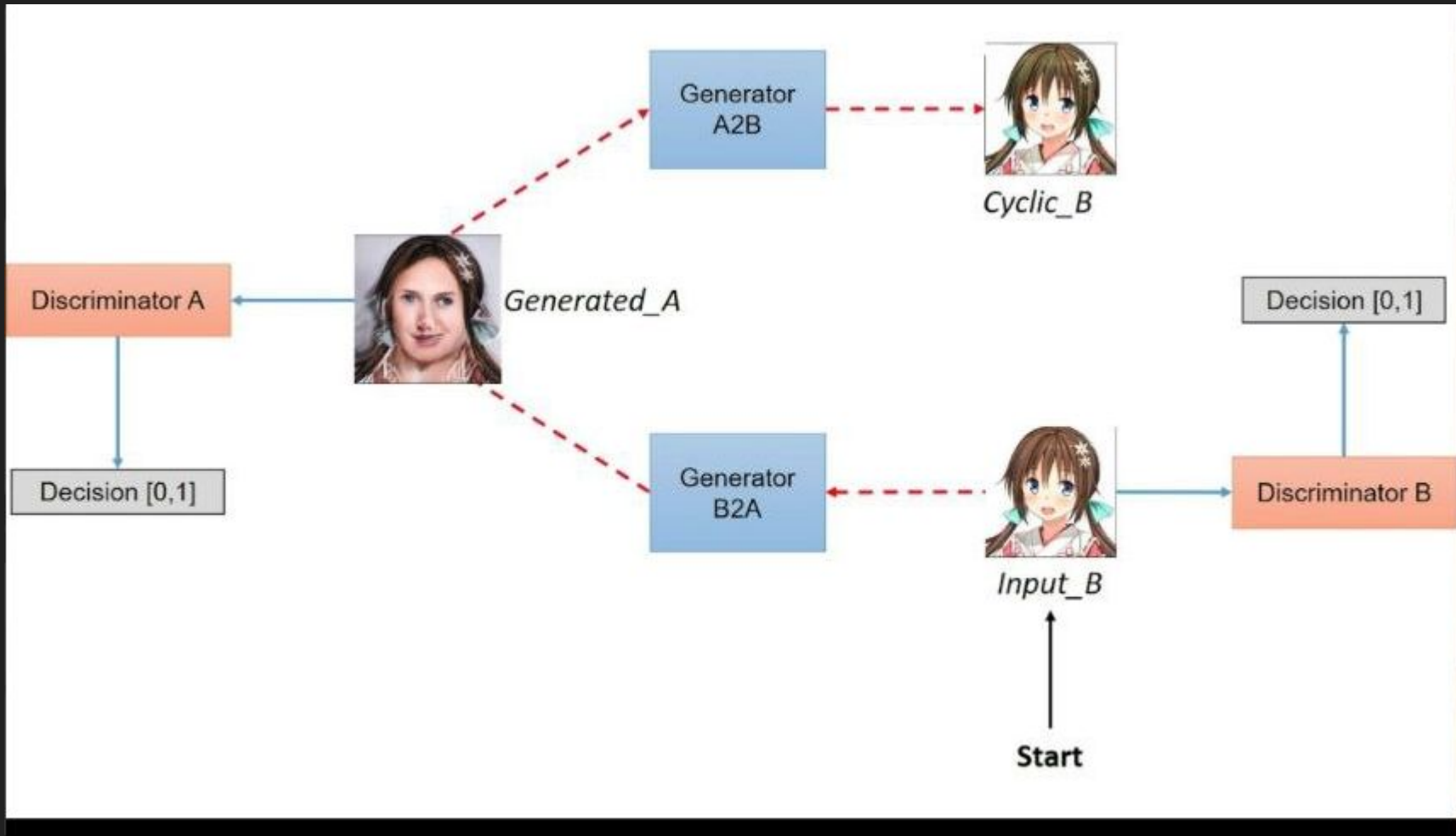


## Циклический переход A2B

Дискриминатор A



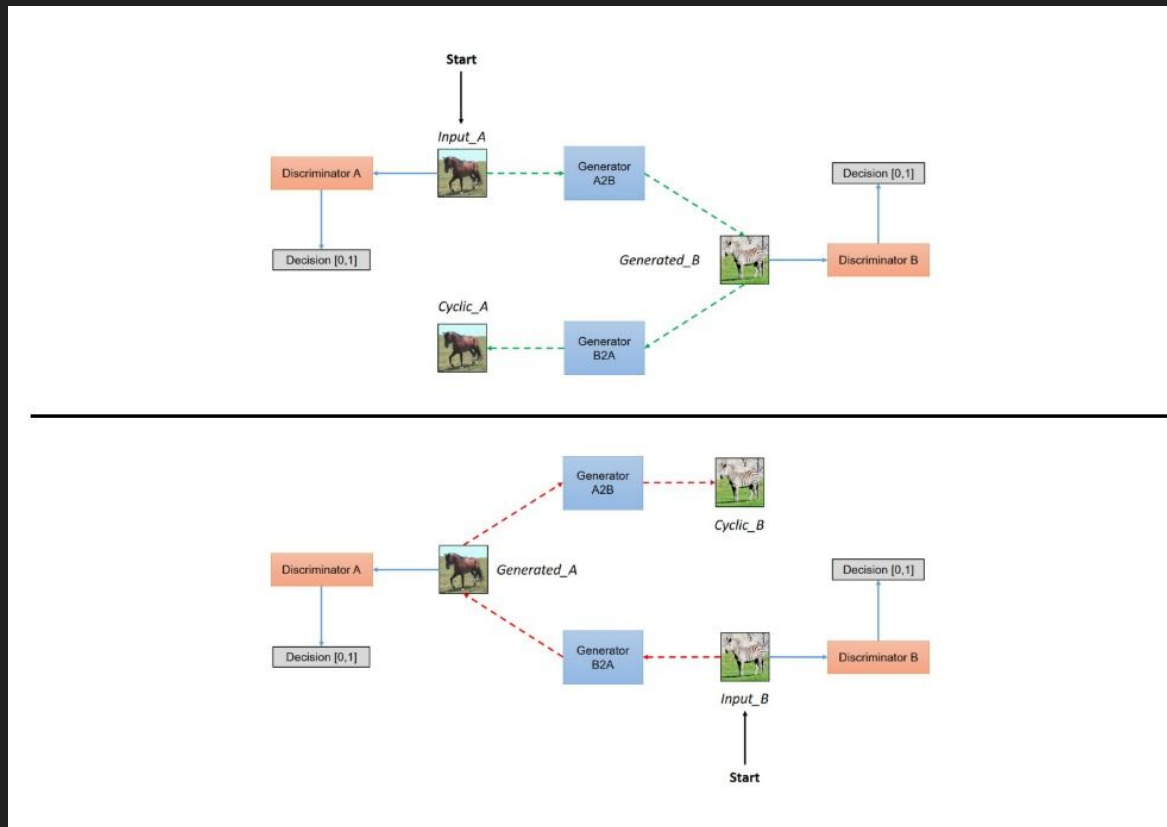
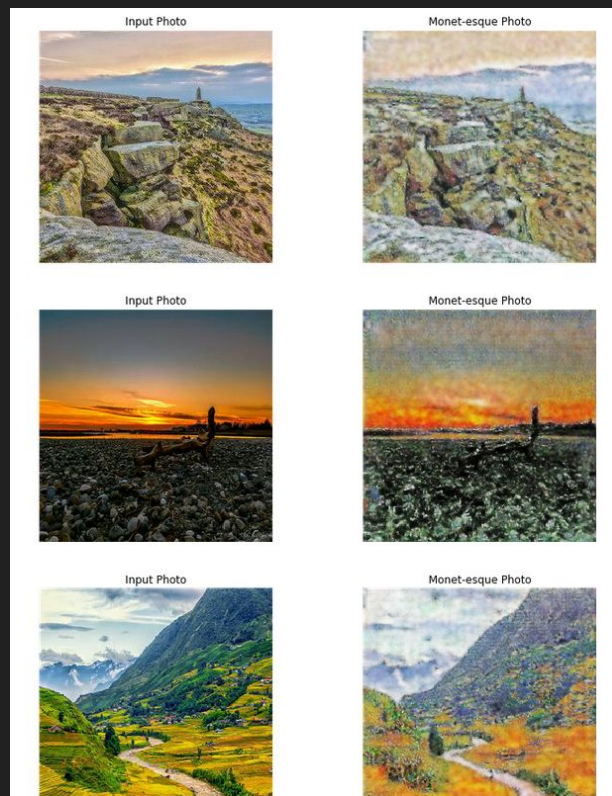
L1 Loss



Succeeded

51.07050

best score solution result RAdam (betas 0.5, 0.999) with SiLU's (tanh for disc) 250 epochs

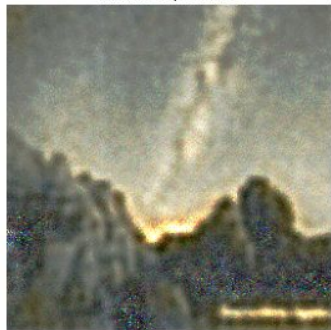


# First CycleGAN solution (leaky relu, batchnorm, without betas for adam)

Input Photo



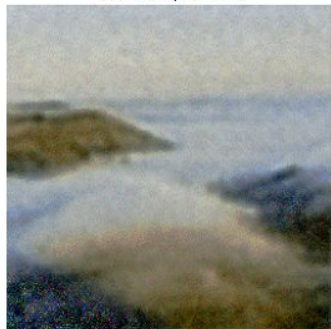
Monet-esque Photo



Input Photo



Monet-esque Photo



**Public Score**

97.57712

# ADAM + SGD 100+100 epochs (without optimized activations)



Input Photo



Monet-esque Photo



Input Photo



Monet-esque Photo



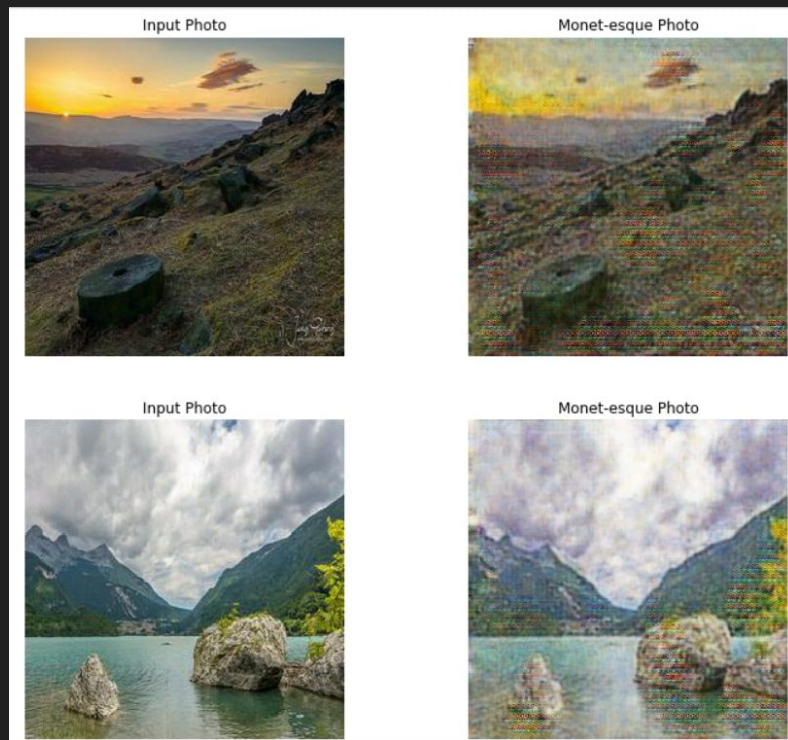
Input Photo



Monet-esque Photo



# ADAM + SGD 100+100 with better activations



# InstanceNorm + different activations ( also add activation before resblock addition)

GOOD RESULTS ONLY ON 30 EPOCHS



Input Photo



Monet-esque Photo



Input Photo




Monet-esque Photo




# But... on 100 epochs we've forgotten “ “ on kaggle

Epoch: (98) | Generator Loss:5.540222 | Discriminator Loss:0.142604

100%  75/75 [01:51<00:00, 1.48s/it, desc\_loss=0.129, gen\_loss=5.96]

Epoch: (99) | Generator Loss:5.509956 | Discriminator Loss:0.138159

100%  75/75 [01:51<00:00, 1.49s/it, desc\_loss=0.144, gen\_loss=5.54]

Epoch: (100) | Generator Loss:5.518247 | Discriminator Loss:0.136401

Epoch: (90) | Generator Loss:5.677043 | Discriminator Loss:0.143393

Epoch: (91) | Generator Loss:5.579003 | Discriminator Loss:0.453186

Epoch: (92) | Generator Loss:5.450772 | Discriminator Loss:0.181949

Epoch: (93) | Generator Loss:5.498578 | Discriminator Loss:0.152772

Epoch: (94) | Generator Loss:5.591769 | Discriminator Loss:0.135696

Epoch: (95) | Generator Loss:5.603216 | Discriminator Loss:0.125866

Epoch: (96) | Generator Loss:5.662980 | Discriminator Loss:0.138459

Epoch: (97) | Generator Loss:5.521772 | Discriminator Loss:0.157073

Epoch: (98) | Generator Loss:5.588758 | Discriminator Loss:0.271601

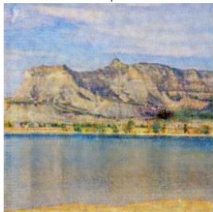
Epoch: (99) | Generator Loss:5.494283 | Discriminator Loss:0.195191

Epoch: (100) | Generator Loss:5.451146 | Discriminator Loss:0.191412

Input Photo



Monet-esque Photo



Input Photo



Monet-esque Photo



In [29]:

```
ph_ds = PhotoDataset('../input/gan-getting-started/photo_jpg/')
```

```
File "/tmp/ipykernel_23/3513488361.py", line 1
```

```
ph_ds = PhotoDataset('../input/gan-getting-started/photo_jpg/')
```

```
SyntaxError: EOL while scanning string literal
```

In [ ]:

```
ph_dl = DataLoader(ph_ds, batch_size=1, pin_memory=True)
```



## RADAM

CLASS `torch.optim.RAdam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0)` [SOURCE]

Implements RAdam algorithm.

**input** :  $\gamma$  (lr),  $\beta_1, \beta_2$  (betas),  $\theta_0$  (params),  $f(\theta)$  (objective),  $\lambda$  (weightdecay),  $\epsilon$  (epsilon)  
**initialize** :  $m_0 \leftarrow 0$  ( first moment),  $v_0 \leftarrow 0$  ( second moment),  
 $\rho_\infty \leftarrow 2/(1 - \beta_2) - 1$

**for**  $t = 1$  **to** ... **do**

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$

**if**  $\lambda \neq 0$

$g_t \leftarrow g_t + \lambda \theta_{t-1}$

$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$

$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$

$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\rho_t \leftarrow \rho_\infty - 2t\beta_2^t / (1 - \beta_2^t)$

**if**  $\rho_t > 5$

$l_t \leftarrow \sqrt{(1 - \beta_2^t) / (v_t + \epsilon)}$

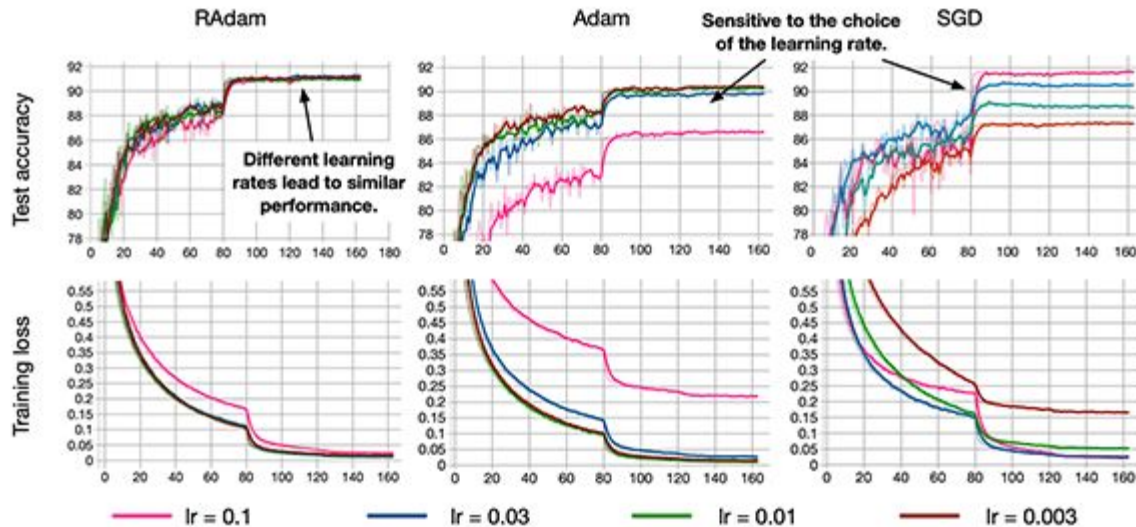
$r_t \leftarrow \sqrt{\frac{(\rho_t - 4)(\rho_t - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_t}}$

$\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m}_t r_t l_t$

**else**

$\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m}_t$

**return**  $\theta_t$



We found that adam/radam with `betas=(0.5, 0.999)` shows better results (from other kaggle notebooks)

```
# -----
monet_desc_real_loss = self.mse_loss(monet_desc_real, real)
monet_desc_fake_loss = self.mse_loss(monet_desc_fake, fake)
photo_desc_real_loss = self.mse_loss(photo_desc_real, real)
photo_desc_fake_loss = self.mse_loss(photo_desc_fake, fake)
```

```
cycle_loss_monet = self.l1_loss(cycl_monet, monet_img) * self.lambda
cycle_loss_photo = self.l1_loss(cycl_photo, photo_img) * self.lambda
```

Epoch: (223)	Generator Loss:5.134601	Discriminator Loss:0.051297
Epoch: (224)	Generator Loss:5.119832	Discriminator Loss:0.060068
Epoch: (225)	Generator Loss:5.076763	Discriminator Loss:0.053184
Epoch: (226)	Generator Loss:5.112150	Discriminator Loss:0.048400
Epoch: (227)	Generator Loss:5.142702	Discriminator Loss:0.046397
Epoch: (228)	Generator Loss:5.145940	Discriminator Loss:0.047137
Epoch: (229)	Generator Loss:5.188370	Discriminator Loss:0.050933
Epoch: (230)	Generator Loss:5.082115	Discriminator Loss:0.051382
Epoch: (231)	Generator Loss:5.126751	Discriminator Loss:0.045191
Epoch: (232)	Generator Loss:5.179132	Discriminator Loss:0.041897
Epoch: (233)	Generator Loss:5.106264	Discriminator Loss:0.044635
Epoch: (234)	Generator Loss:5.127832	Discriminator Loss:0.044148
Epoch: (235)	Generator Loss:5.114143	Discriminator Loss:0.038463
Epoch: (236)	Generator Loss:5.167543	Discriminator Loss:0.046337
Epoch: (237)	Generator Loss:5.131823	Discriminator Loss:0.044436
Epoch: (238)	Generator Loss:5.130623	Discriminator Loss:0.038897
Epoch: (239)	Generator Loss:5.136845	Discriminator Loss:0.046262
Epoch: (240)	Generator Loss:5.167635	Discriminator Loss:0.037352
Epoch: (241)	Generator Loss:5.246021	Discriminator Loss:0.039936
Epoch: (242)	Generator Loss:5.195492	Discriminator Loss:0.036199
Epoch: (243)	Generator Loss:5.196167	Discriminator Loss:0.033596
Epoch: (244)	Generator Loss:5.154130	Discriminator Loss:0.033648
Epoch: (245)	Generator Loss:5.142621	Discriminator Loss:0.032786
Epoch: (246)	Generator Loss:5.106339	Discriminator Loss:0.037740
Epoch: (247)	Generator Loss:5.132276	Discriminator Loss:0.034971
Epoch: (248)	Generator Loss:5.099536	Discriminator Loss:0.031717
Epoch: (249)	Generator Loss:5.131913	Discriminator Loss:0.029916
Epoch: (250)	Generator Loss:5.192141	Discriminator Loss:0.033124

### Batch normalization

$$y_{tijk} = \frac{x_{tijk} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}, \quad \mu_i = \frac{1}{HWT} \sum_{t=1}^T \sum_{l=1}^W \sum_{m=1}^H x_{tilm}, \quad \sigma_i^2 = \frac{1}{HWT} \sum_{t=1}^T \sum_{l=1}^W \sum_{m=1}^H (x_{tilm} - \mu_i)^2.$$

### Instance normalization

$$y_{tijk} = \frac{x_{tijk} - \mu_{ti}}{\sqrt{\sigma_{ti}^2 + \epsilon}}, \quad \mu_{ti} = \frac{1}{HW} \sum_{l=1}^W \sum_{m=1}^H x_{tilm}, \quad \sigma_{ti}^2 = \frac{1}{HW} \sum_{l=1}^W \sum_{m=1}^H (x_{tilm} - \mu_{ti})^2.$$

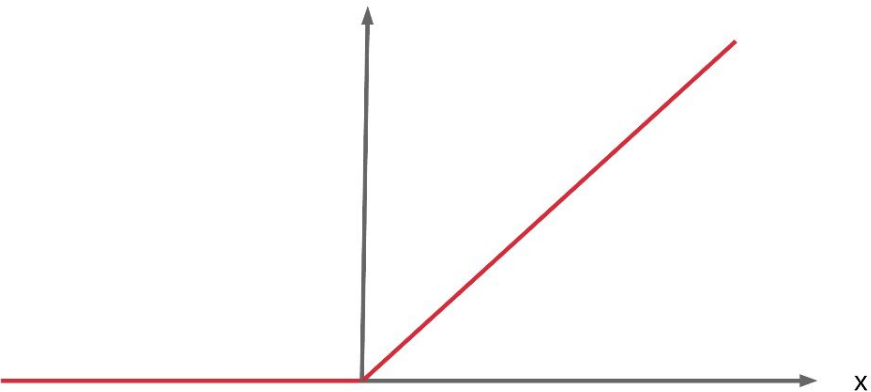
Batch version normalizes all images across the batch and spatial locations (in the CNN case, in the ordinary case it's different); instance version normalizes each element of the batch independently, i.e., across spatial locations only.

batch normalization adds extra noise to the training, because the result for a particular instance depends on the neighbor instances

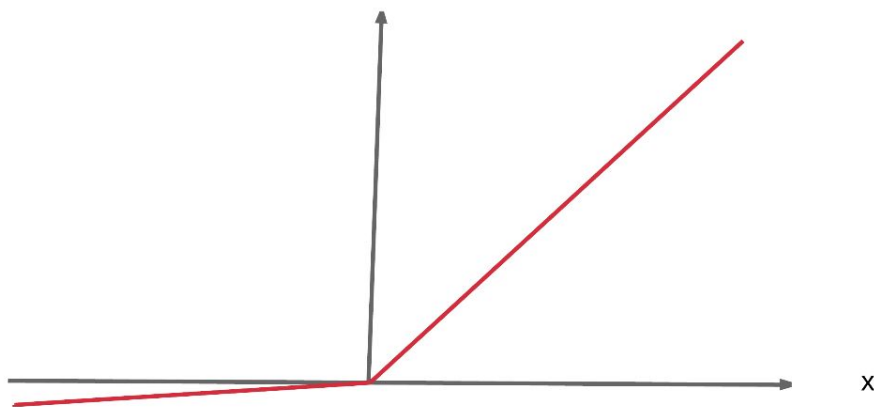
we don't need that extra noise (also the results from other people show us, that InstanceNorm works better

# We used leakyrelu

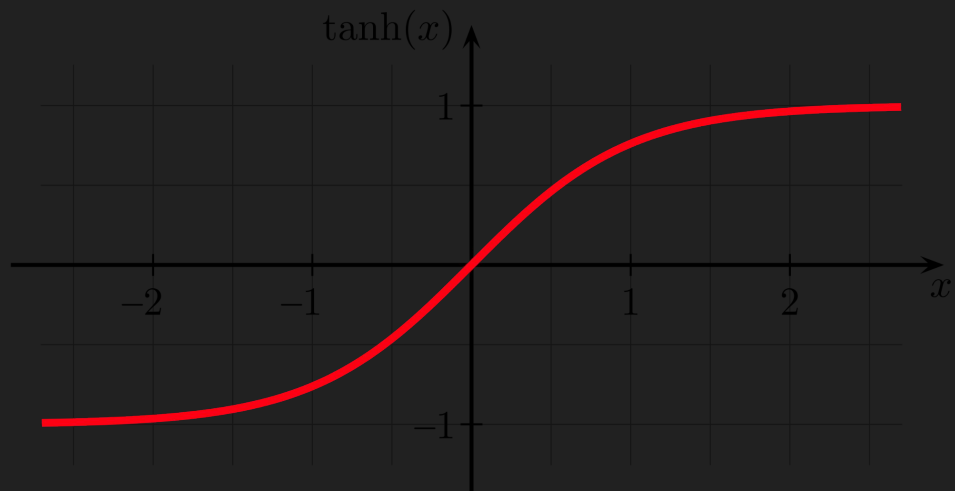
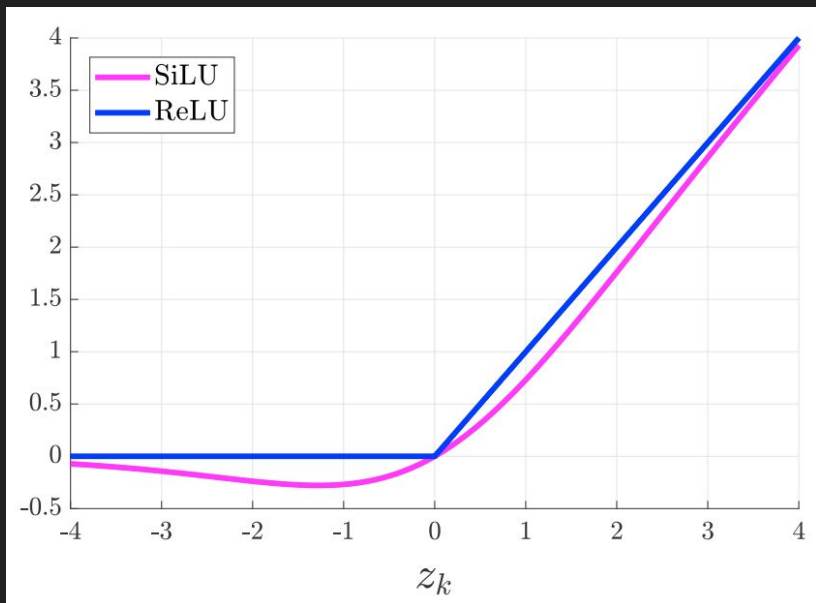
$\text{ReLU}(x)$



$\text{LeakyReLU}(x)$



# SiLU and TANH





From the DCGAN paper [Radford et al. <https://arxiv.org/pdf/1511.06434.pdf>]...

10



"The ReLU activation (Nair & Hinton, 2010) is used in the generator with the exception of the output layer which uses the Tanh function. We observed that using a bounded activation allowed the model to learn more quickly to saturate and cover the color space of the training distribution. Within the discriminator we found the leaky rectified activation (Maas et al., 2013) (Xu et al., 2015) to work well, especially for higher resolution modeling. This is in contrast to the original GAN paper, which used the maxout activation (Goodfellow et al., 2013)."

It could be that the symmetry of tanh is an advantage here, since the network should be treating darker colours and lighter colours in a symmetric way.

Share Improve this answer Follow

edited Mar 20, 2017 at 16:56

answered Mar 20, 2017 at 16:32



Ben Carr

133 • 1 • 5

[cyclegan\\_monet](#)  
(version 6/9)

[Roman Derunets](#)

cyclegan\_monet | Version RoRoNet (LeakyReLU, BS=1, sheduler - exponential)

Succeeded 97.57712

[notebook9758630d4d](#)  
(version 3/3)

[Roman Derunets](#)

sgd + radam


Succeeded 95.15319







[cyclegan\\_monet](#)  
ver8 (version 8/9)

[Roman Derunets](#)

LambdaLR schedule, Adam with betas, 30 epochs

Succeeded 79.19065

133 **Rodion Shkokov**  68.42106 1 2m

 Your First Entry!  
Welcome to the leaderboard!     

[30 radam instance](#)

30 epochs radam instance (version 2/2)

[Roman Derunets](#)

Notebook 30 radam instance | 30 epochs radam instance

Succeeded 57.34186

[notebooka700af2b6f](#)

RADAM250 (version 2/2)

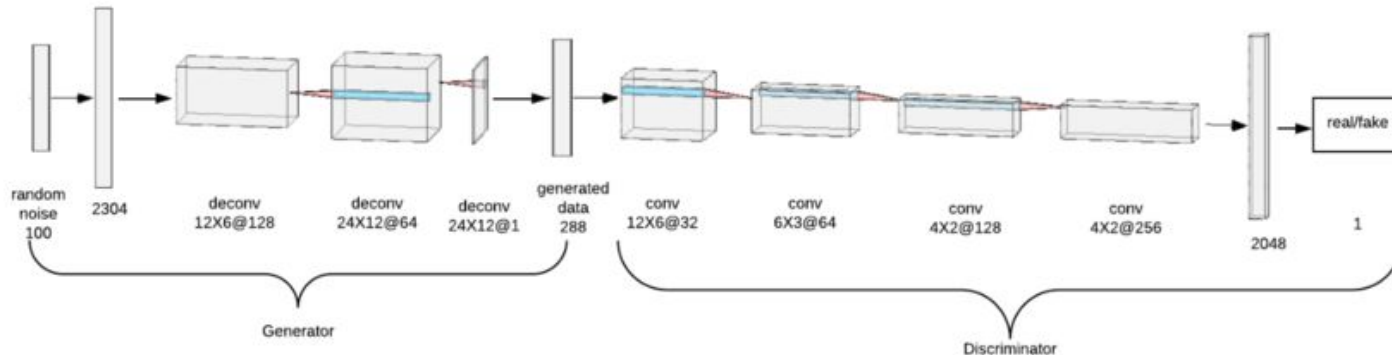
[Roman Derunets](#)

Notebook | RADAM250

Succeeded 51.07050

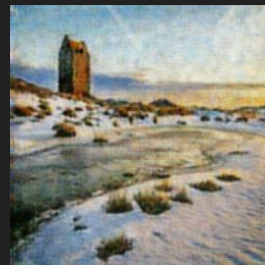
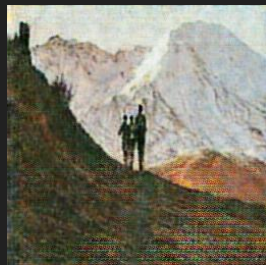
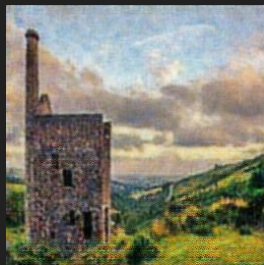
A summary of some of the more actionable tips is provided below.

- Normalize inputs to the range  $[-1, 1]$  and use tanh in the generator output.
- Flip the labels and loss function when training the generator.
- Sample Gaussian random numbers as input to the generator.
- Use mini batches of all real or all fake for calculating batch norm statistics.
- Use Leaky ReLU in the generator and discriminator.
- Use Average pooling and stride for downsampling; use ConvTranspose2D and stride for upsampling.
- Use label smoothing in the discriminator, with small random noise.
- Add random noise to the labels in the discriminator.
- Use DCGAN architecture, unless you have a good reason not to.
- A loss of 0.0 in the discriminator is a failure mode.
- If loss of the generator steadily decreases, it is likely fooling the discriminator with garbage images.
- Use labels if you have them.
- Add noise to inputs to the discriminator and decay the noise over time.
- Use dropout of 50 percent during train and generation.





Try to guess where is the real Monet and Fakes



# Полезные ссылки

- <https://towardsdatascience.com/overview-of-cyclegan-architecture-and-training-afee31612a2f>
- <https://towardsdatascience.com/cyclegan-learning-to-translate-images-without-paired-training-data-5b4e93862c8d>
- [https://youtu.be/T-IBMrjZ3\\_0](https://youtu.be/T-IBMrjZ3_0)
- <https://www.kaggle.com/code/amyjang/monet-cyclegan-tutorial>
- <https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix>
- <https://www.youtube.com/watch?v=s8F4rjiJsC4>

## Failure cases



67

MMM



51.07050

8

2h



Your Best Entry!

Your submission scored 57.34186, which is not an improvement of your previous score. Keep trying!