

Generative Adversarial Networks: Wasserstein GAN

Mehmet Can Demir

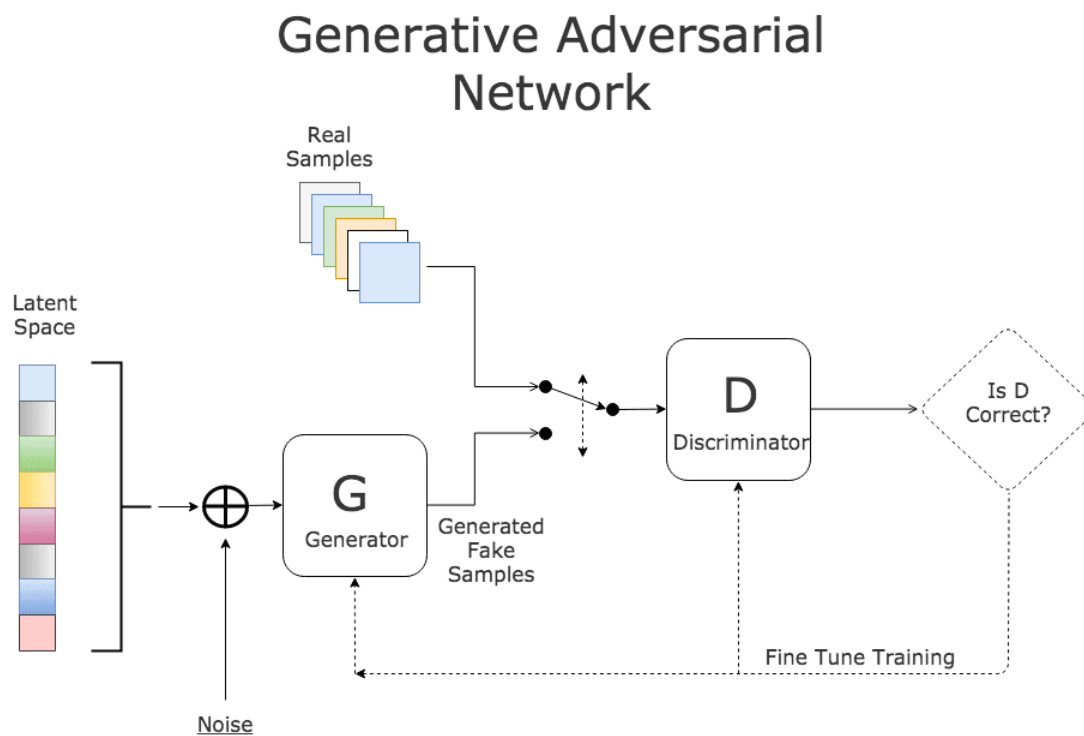
Introduction

Generative Adversarial Networks (GANs) became more popular after NVIDIA's announcement of Progressively Growing GAN (ProGAN) [1] with undeniable improvements and generation of high resolution human faces. ProGAN outperformed almost all the previous GAN types such as DCGAN, WGAN, WGAN-GP.

The main idea of a GAN model is to create a probability distribution that is as close as the probability distribution of the given training data. GAN is split in two parts:

1. Generator
2. Discriminator

The generator is responsible to generate the targeted data given a noise variable input z , meanwhile the discriminator works as a critic, and is responsible to differentiate a generated data with a real one. Both of these networks play a min-max game where one is trying to outsmart the other. This game between these models, motivates both of them to improve their functionalities.



Generative Adversarial Networks: Wasserstein GAN

How do they work?

To calculate the probabilities, two different probability distributions are used:

1. Kullback-Leibler Divergence
2. Jensen-Shannon Divergence

Kullback-Leibler (KL) divergence measures how one probability distribution p diverges from a second expected probability distribution q .

$$D_{KL}(p||q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx$$

It is noticeable that in cases where $p(x)$ is close to zero, but $q(x)$ is significantly non-zero, the q 's effect is disregarded. It could cause buggy results when we just want to measure the similarity between two equally important distributions.

Jensen-Shannon (JS) divergence is another measure of similarity between two probability distributions, bounded by $[0, 1]$. JS divergence is more symmetric and smooth than KL divergence.

$$D_{JS}(p||q) = \frac{1}{2} D_{KL}(p||\frac{p+q}{2}) + \frac{1}{2} D_{KL}(q||\frac{p+q}{2})$$

Discriminator's decisions over real data must be made accurately by maximizing $E_{x \sim p_r(x)} [\log D(x)]$. Meanwhile, given a fake sample $G(z)$, $z \sim p_z(z)$, the discriminator is expected to output a probability, $D(G(z))$, close to zero by maximizing $E_{z \sim p_z(z)} [\log (1 - D(G(z)))]$.

On the other hand, the generator is trained to increase the chances of D producing a high probability for a fake example, thus to minimize $E_{z \sim p_z(z)} [\log (1 - D(G(z)))]$.

With this game, an optimized loss can be called as min-max GAN loss function, as simplified below:

Generative Adversarial Networks: Wasserstein GAN

$$\begin{aligned}\min_G \max_D L(D, G) &= \mathbb{E}_{x \sim p_r(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))] \\ &= \mathbb{E}_{x \sim p_r(x)} [\log D(x)] + \mathbb{E}_{x \sim p_g(x)} [\log(1 - D(x))]\end{aligned}$$

While the discriminator is trained, it classifies both the real data and the generated data. It penalizes itself for misclassifying a real instance as fake, or a fake instance as real, by maximizing the function:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D \left(x^{(i)} \right) + \log \left(1 - D \left(G \left(z^{(i)} \right) \right) \right) \right]$$

While the generator is trained, it samples random noise and produces an output from that noise. The output then goes through the discriminator and gets classified as either “Real” or “Fake” based on the ability of the discriminator to tell one from the other. The generator loss is then calculated from the discriminator’s classification; it gets rewarded if it successfully fools the discriminator, and gets penalized otherwise.

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log \left(1 - D \left(G \left(z^{(i)} \right) \right) \right)$$

Note: This work is aimed to understand and demonstrate how Wasserstein GAN (WGAN) [2] model architecture works and applying Gradient Penalty (GP) to improve the performance of the trained generator.

From GAN to Wasserstein GAN(WGAN)

WGAN is one of the most powerful alternatives to the original GAN loss. It tackles the problems of Mode Collapse and Vanishing Gradient.

Generative Adversarial Networks: Wasserstein GAN

There are also three things to consider which are differentiates WGAN from a standard GAN model:

1. Since the discriminator doesn't actually classify the data strictly as real or fake, it is called **Critique** $f(x)$ instead.
2. Wasserstein GAN's name comes from its own probability distance formulation: Wasserstein Distance

In WGAN implementation, the activation of the output layer of the discriminator is changed from sigmoid to a linear one, so the discriminator can give out a score instead of a probability associated with data distribution.

$$\nabla_w \frac{1}{m} \sum_{i=1}^m [f(x^{(i)}) - f(G(z^{(i)}))]$$

Generator is acts the same as in standard GAN, it takes a latent noise z and produces data with learned probability distributions.

$$\nabla_{\theta} \frac{1}{m} \sum_{i=1}^m f(G(z^{(i)}))$$

WGAN - Gradient Penalty (WGAN-GP)

The idea of Gradient Penalty is to enforce a constraint such that the gradients of the critic's output with respect to the inputs to have unit norm. The default WGAN loss (or value) function was:

$$\min_G \max_{\|f\|_L \leq 1} \mathbb{E}[f(\mathbf{x})] - \mathbb{E}[f(\tilde{\mathbf{x}})]$$

Generative Adversarial Networks: Wasserstein GAN

The issues with WGAN arise mainly because of the weight clipping method used to enforce Lipschitz continuity on the critic. WGAN-GP replaces weight clipping with a constraint on the gradient norm of the critic to enforce Lipschitz continuity and allows for more stable training of the network. Thus, with WGAN-GP, the new loss is described as below, where λ , the penalty coefficient is used to weight the gradient penalty term.:

$$\mathcal{L} = \mathbb{E}_{\mathbf{x} \sim \mathbb{P}_r} [f(\mathbf{x})] - \mathbb{E}_{\tilde{\mathbf{x}} \sim \mathbb{P}_g} [f(\tilde{\mathbf{x}})] + \lambda \mathbb{E}_{\hat{\mathbf{x}} \sim \mathbb{P}_{\hat{x}}} [(||\nabla_{\hat{\mathbf{x}}} f(\hat{\mathbf{x}})||_2 - 1)^2]$$

Data

The data used is called CatFaces, which consists of over ~15.000 images of cat faces (64x64). The preparation of data is followed by 3 steps:

1. Downloading the full dataset from the source.
2. Preparing custom data which consisted of 506 collected images of cat faces.
3. Merging the custom data with the main one.

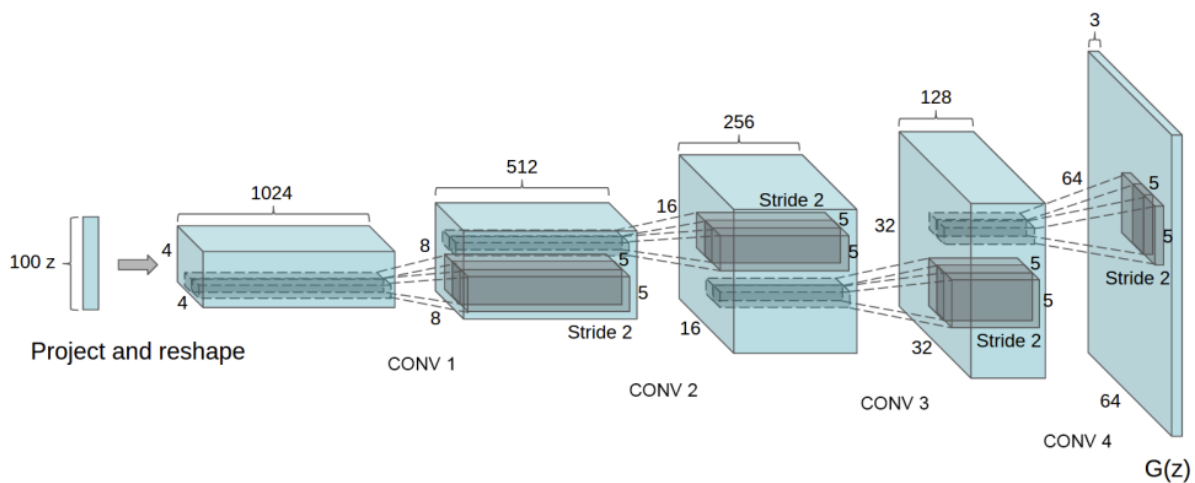


Generative Adversarial Networks: Wasserstein GAN

Model Architecture

Main Idea

The main idea of the used architecture was based on Deep Convolutional GANs. At the start, a 100 dimensional uniform distribution Z is initiated and projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions then convert this high level representation into a 64x64 pixel image without using any fully connected or pooling layers.[4]



Description

1. Main Libraries

- a. **PyTorch:** Pytorch is an open source Machine Learning / Deep Learning library based on the Torch library. It is a popular library that is used (and being used) for many applications such as computer vision, natural language processing, image processing, object detection and so on. It is primarily developed by Facebook's AI Research lab.
- b. **Torchvision:** Torchvision is a library for Computer Vision that goes hand in hand with PyTorch. It has many utilities for efficient image and video transformations, commonly used pre-trained models, and some datasets.

Generative Adversarial Networks: Wasserstein GAN

- c. **TorchInfo:** TorchInfo provides information complementary to what is provided by `print(model)` in PyTorch, it is similar to TensorFlow's `model.summary()` API to view the visualization of the model, which is incredibly helpful while debugging a network.
- d. **PyYAML:** PyYAML is a data serialization format designed for human readability and interaction with scripting languages is a YAML parser and emitter for Python. It features a complete YAML 1.1 parser, Unicode support, pickle support, capable extension API, and sensible error messages. PyYAML supports standard YAML tags and provides Python-specific tags that allow it to represent an arbitrary Python object.
- e. **Matplotlib:** Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt or GTK.
- f. **Tensorboard:** Tensorboard provides the visualization and tooling needed for machine learning experimentation such as tracking and visualization metrics (loss, accuracy, etc.), visualizing the model graph, viewing histograms of weights/biases/tensors, displaying images, projecting embeddings.

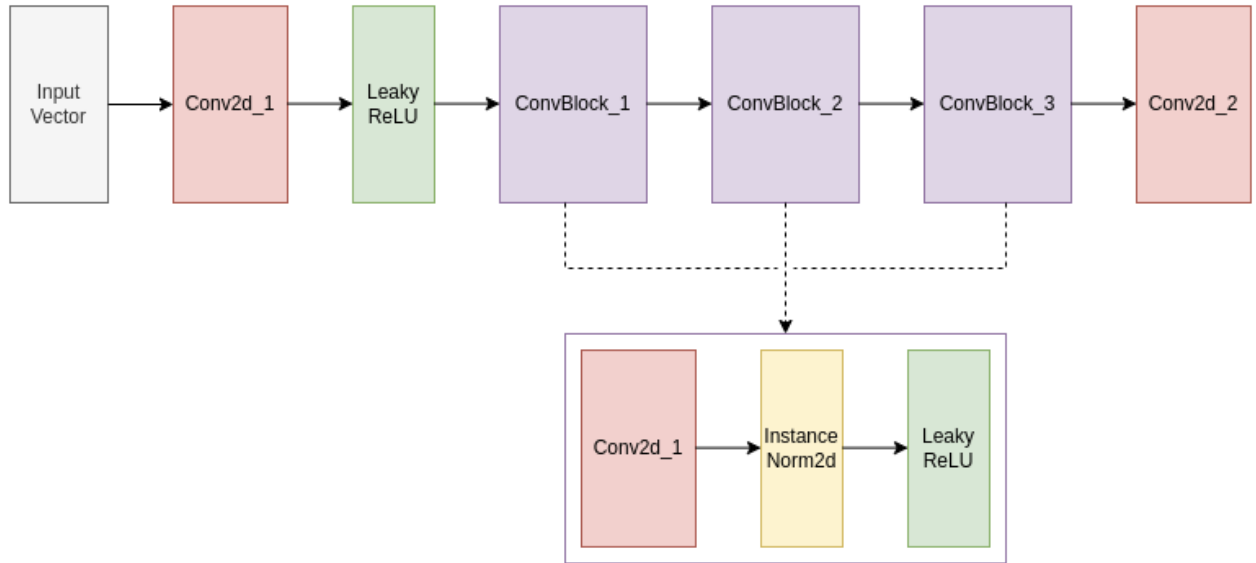
2. Modules and Files

- a. **train.py:** Consists the code which is responsible for reading the training configurations, preparing the data that will be trained on, argument parsing, saving model (with checkpoints in every epoch), writing logs on the tensorboard and to train the model.
- b. **utils.py:** Controls the training/ folder structure in order to prevent unintentional training overwriting, and consists of the code that is responsible to generate gradient penalties.
- c. **run.py:** Responsible for evaluating the pre-trained models by using matplotlib figures.
- d. **train.config.yaml:** Includes the training hyperparameters with some extra options (such as device and data path/name).

Generative Adversarial Networks: Wasserstein GAN

3. Discriminator - 64x64

A deeper look to the discriminator's architecture is shown below:



InputVector: Input vector, the required shape of the input data which the corresponding model accepts. In this architecture, the inputs must be in the shape of $N \times 3 \times 64 \times 64$.

Conv2d: Convolutional Layer, is the layer where convolution operation is applied. Then, passes it's outputs to the next layer. At first, the input data (in our case, an image) is padded with zeros, while in the second step the kernel is placed on the padded input and slid across generating the output pixels as dot products of the kernel and the overlapped input region.

LeakyReLU: Leaky Rectified Linear Unit, or Leaky ReLU, is a type of activation function based on a ReLU, but it has a small slope for negative values instead of a flat slope. The slope coefficient is determined before training, i.e. it is not learnt during training. This type of activation function is popular in tasks where we may suffer from sparse gradients, for example training *Generative Adversarial Networks*.

ConvBlock: Used for repeating groups of layers such as “Convolutional, Instance Normalization, Leaky ReLU”.

Generative Adversarial Networks: Wasserstein GAN

InstanceNorm2d: Instance Normalization tells us that it operates on a single sample, while Batch Normalization operates on one channel over all training samples in the mini-batch.

```
class Discriminator(nn.Module):
    def __init__(self, channels_img, features_d):
        super(Discriminator, self).__init__()
        self.disc = nn.Sequential(
            nn.Conv2d(channels_img, features_d, kernel_size=4, stride=2,
padding=1),
            nn.LeakyReLU(0.2),
            self._block(features_d, features_d * 2, 4, 2, 1),
            self._block(features_d * 2, features_d * 4, 4, 2, 1),
            self._block(features_d * 4, features_d * 8, 4, 2, 1),
            nn.Conv2d(features_d * 8, 1, kernel_size=4, stride=2,
padding=0)
        )

    def _block(self, in_channels, out_channels, kernel_size, stride,
padding):
        return nn.Sequential(
            nn.Conv2d(
                in_channels,
                out_channels,
                kernel_size,
                stride,
                padding,
                bias=False,
            ),
            nn.InstanceNorm2d(out_channels, affine=True),
            nn.LeakyReLU(0.2),
        )

    def forward(self, x):
        return self.disc(x)
```

```
=====
Total params: 44,649
Trainable params: 44,649
Non-trainable params: 0
Total mult-adds (M): 15.80
=====
```

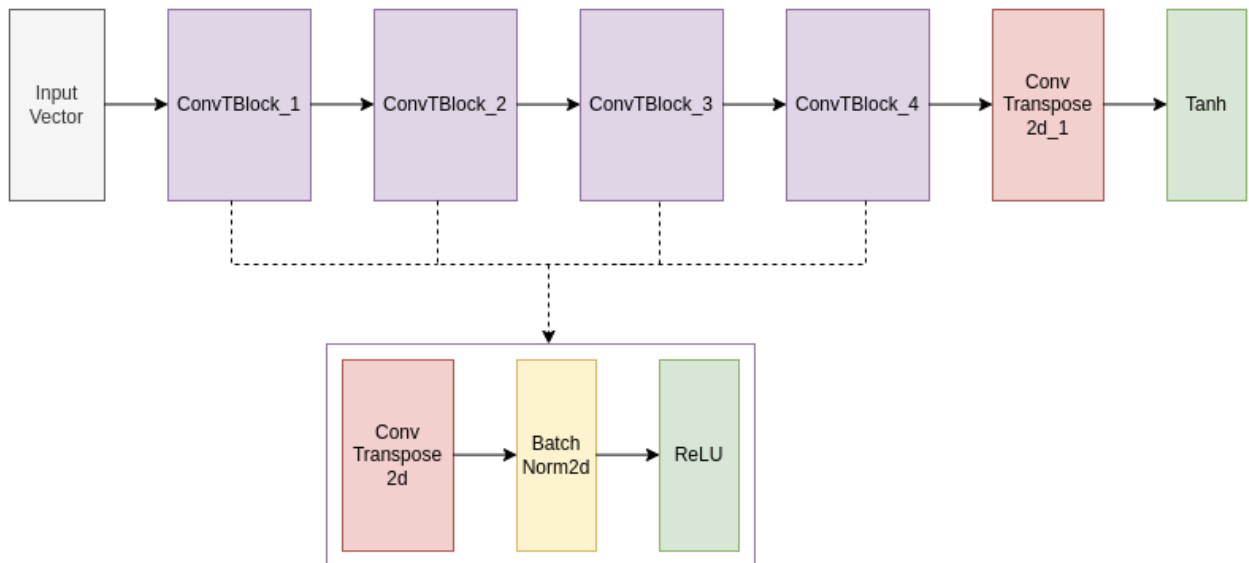
```
Input size (MB): 0.39
Forward/backward pass size (MB): 1.44
```

Generative Adversarial Networks: Wasserstein GAN

Params size (MB): 0.18
Estimated Total Size (MB): 2.01

4. Generator - 64x64

A deeper look to the dicriminator's architecture is shown below:



InputVector: Input vector, the required shape of the input data which the corresponding model accepts. In this architecture, the inputs must be in the shape of $N \times 1 \times 1 \times 1$.

ConvTranspose2d: The transposed convolutional layer is similar to the deconvolutional layer in the sense. It outsamples to generate an output feature map that has a spatial dimension greater than that of the input feature map. Just like the standard convolutional layer, the transposed convolutional layer is also defined by the padding and stride. These values of padding and stride are the one that hypothetically was carried out on the output to generate the input.

ReLU: Rectified Linear Unit, or ReLU for short. It is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks since it makes the training phase easier and faster. Thus, in most of the cases it achieves better performance.

Generative Adversarial Networks: Wasserstein GAN

Tanh: Tanh is a hyperbolic tangent activation function. It is very similar to the sigmoid activation function since both have the same shape. It takes any real value as input, and outputs values in the range of -1 to 1.

ConvTBlock: Used for repeating groups of layers such as “Convolutional Transpose2d, Batch Normalization, ReLU”.

InstanceNorm2d: Instance Normalization tells us that it operates on a single sample, while Batch Normalization operates on one channel over all training samples in the mini-batch.

```
class Generator(nn.Module):
    def __init__(self, z_dim, channels_img, features_g):
        super(Generator, self).__init__()
        self.gen = nn.Sequential(
            self._block(z_dim, features_g * 16, 4, 1, 0),
            self._block(features_g * 16, features_g * 8, 4, 2, 1),
            self._block(features_g * 8, features_g * 4, 4, 2, 1),
            self._block(features_g * 4, features_g * 2, 4, 2, 1),
            nn.ConvTranspose2d(
                features_g * 2, channels_img, kernel_size=4, stride=2,
padding=1
            ),
            nn.Tanh(),
        )

    def _block(self, in_channels, out_channels, kernel_size, stride,
padding):
        return nn.Sequential(
            nn.ConvTranspose2d(
                in_channels,
                out_channels,
                kernel_size,
                stride,
                padding,
                bias=False,
            ),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
        )

    def forward(self, x):
        return self.gen(x)
```

Generative Adversarial Networks: Wasserstein GAN

```
=====
Total params: 378,083
Trainable params: 378,083
Non-trainable params: 0
Total mult-adds (M): 252.81
=====
```

```
=====
Input size (MB): 0.00
Forward/backward pass size (MB): 4.72
Params size (MB): 1.51
Estimated Total Size (MB): 6.23
=====
```

5. Runtime Environment

OS: Ubuntu 21.04

Processor: Intel(R) Core(TM) i5-10200H CPU @ 2.40GHz

Memory: 16 GB

Video Card: NVIDIA GeForce GTX 1650 Ti / 4096 MB

Package Manager: Conda 4.11.0

Python: 3.8.12

Pytorch: 1.10.1

Torchvision: 0.11.2

Matplotlib: 3.5.0

PyYAML: 6.0

6. About Trained Models:

a. CatFaces_1

Training Duration: 3 hours

Inference Time on Evaluation: ~0.13 ms

Hyperparameters: (These parameters are chosen due to Generative Adversarial Networks' high sensitivity to hyperparameters. In the first model, the hyperparameters are the ones that are suggested in papers and communities.)

Generative Adversarial Networks: Wasserstein GAN

NUM_EPOCHS: 15 (even though we can't really determine how many epochs to train, images are 64x64 and not very complicated. That means we can go with a standard number.)

LEARNING_RATE: 1e-4 (suggested learning rate)

BATCH_SIZE: 32

IMAGE_SIZE: 64

CHANNELS_IMG: 3

Z_DIM: 100 (dimensions of the noise)

FEATURES_CRITIC: 64 (channels that will change)

FEATURES_GEN: 64 (channels that will change)

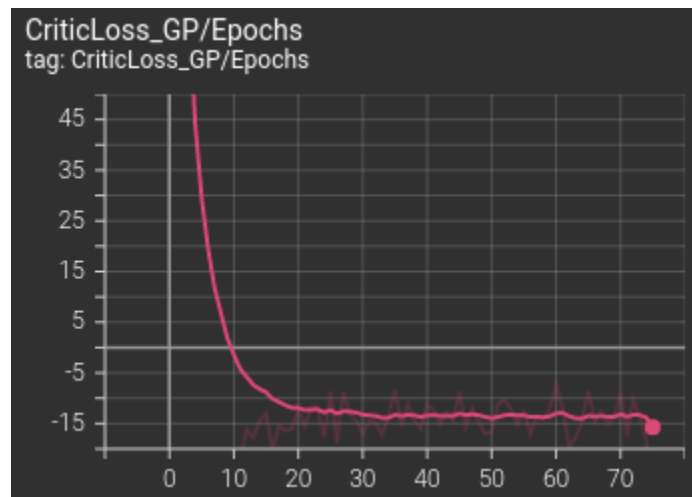
CRITIC_ITERATIONS: 5 (critic training)

LAMBDA_GP: 10 (penalty coefficient)

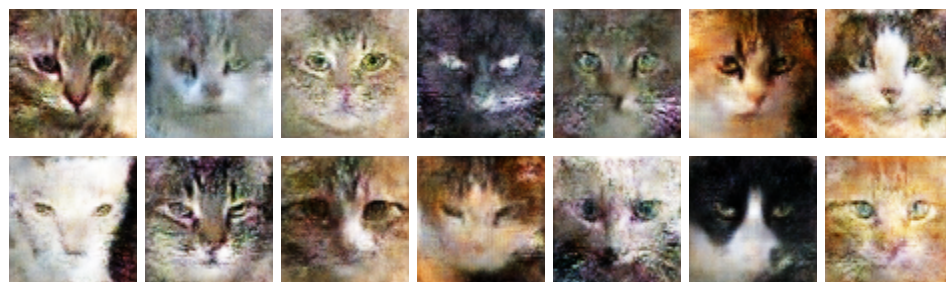
DATA_NAME: CatFaces

DEVICE: cuda

Loss During Training:



Some Generated Examples:



Generative Adversarial Networks: Wasserstein GAN

b. CatFaces_2:

Training Duration: 3 hours

Inference Time on Evaluation: ~0.13 ms

Hyperparameters: (These parameters are chosen due to Generative Adversarial Networks' high sensitivity to hyperparameters. In the first model, the hyperparameters are the ones that are suggested in papers and communities.)

NUM_EPOCHS: 15 (even though we can't really determine how many epochs to train, images are 64x64 and not very complicated. That means we can go with a standard number.)

LEARNING_RATE: $2e-4$ (suggested learning rate)

BATCH_SIZE: 32

IMAGE_SIZE: 64

CHANNELS_IMG: 3

Z_DIM: 100 (dimensions of the noise)

FEATURES_CRITIC: 64 (channels that will change)

FEATURES_GEN: 64 (channels that will change)

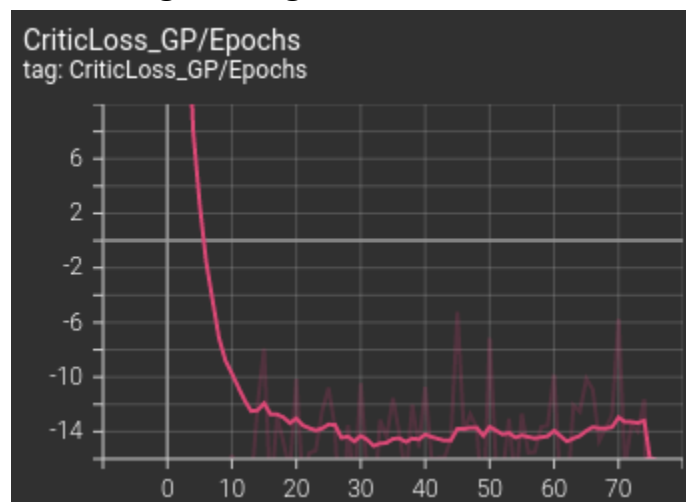
CRITIC_ITERATIONS: 5 (critic training)

LAMBDA_GP: 10 (penalty coefficient)

DATA_NAME: CatFaces

DEVICE: cuda

Loss During Training:



Generative Adversarial Networks: Wasserstein GAN

Some Generated Examples:



c. CatFaces_3:

Training Duration: 6 hours

Inference Time on Evaluation: ~0.13 ms

Hyperparameters: (These parameters are chosen due to Generative Adversarial Networks' high sensitivity to hyperparameters. In the first model, the hyperparameters are the ones that are suggested in papers and communities.)

NUM_EPOCHS: 30 (even though we can't really determine how many epochs to train, images are 64x64 and not very complicated. That means we can go with a standard number.)

LEARNING_RATE: 1e-4 (suggested learning rate)

BATCH_SIZE: 32

IMAGE_SIZE: 64

CHANNELS_IMG: 3

Z_DIM: 100 (dimensions of the noise)

FEATURES_CRITIC: 64 (channels that will change)

FEATURES_GEN: 64 (channels that will change)

CRITIC_ITERATIONS: 5 (critic training)

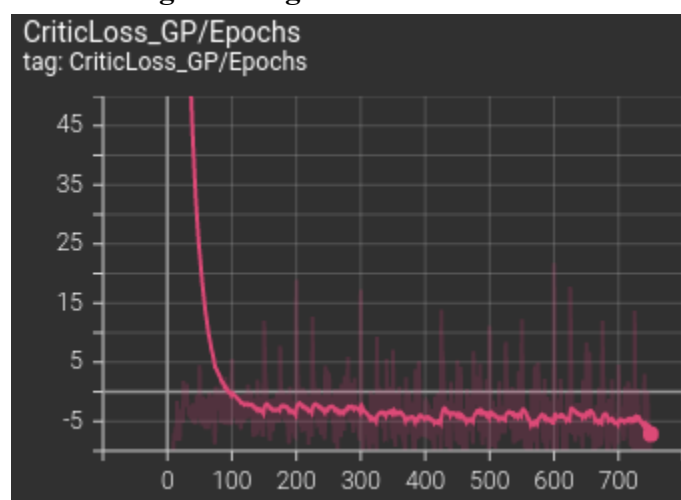
LAMBDA_GP: 10 (penalty coefficient)

DATA_NAME: CatFaces

DEVICE: cuda

Generative Adversarial Networks: Wasserstein GAN

Loss During Training:



Some Generated Examples:



7. Comparison:

	Model_1	Model_2	Model_3
Epoch	15	15	30
Learning Rate	1e-4 (0.001)	2e-4 (0.002)	1e-4 (0.001)
Result			

When we compare those 3 models, we can clearly see training for 30 epochs gives better results. Also, using learning rate 0.001 (as suggested) gives better results than using learning rate of 0.002 (it can be clearly seen that the shape of the cat is not quite accurate).

Generative Adversarial Networks: Wasserstein GAN

Training

1. Preparation

Both Generator and Discriminator are created and initialized with weights.

```
gen = Generator(Z_DIM, CHANNELS_IMG, FEATURES_GEN).to(DEVICE)
critic = Discriminator(CHANNELS_IMG, FEATURES_CRITIC).to(DEVICE)

initialize_weights(gen)
initialize_weights(critic)
```

Optimizers for both Generator and Discriminator are created.

```
opt_gen = optim.Adam(gen.parameters(), lr=LEARNING_RATE, betas=(0.0, 0.9))
opt_critic = optim.Adam(critic.parameters(), lr=LEARNING_RATE, betas=(0.0, 0.9))
```

Set the models to train mode so the layers which behave differently on the train and test procedures know what is going on and hence can behave accordingly.

```
gen.train()
critic.train()
```

2. Training Discriminator

```
for epoch in range(NUM_EPOCHS):

    checkpoint = {
        'state_dict': gen.state_dict(),
        'optimizer': opt_gen.state_dict()
    }

    save_checkpoint(checkpoint, f'{train_dir}/model/catfaces_checkpoint.pth.tar')
```

Get the batch id (to name the images that are generated) and real images from the batch (“_” is the label, which we don't need since the training is unsupervised).

```
for batch_idx, (real, _) in enumerate(loader):
    real = real.to(DEVICE)
    cur_batch_size = real.shape[0]
```

Create a random noise and a fake image using the Generator. Then, put both fake and real images to the Discriminator (which acts like a critic). These values will be used to calculate loss.

Generative Adversarial Networks: Wasserstein GAN

```
# train disc / critic
for _ in range(CRITIC_ITERATIONS):
    noise = torch.randn(cur_batch_size, Z_DIM, 1, 1).to(DEVICE)
    fake = gen(noise)

    critic_real = critic(real).reshape(-1)
    critic_fake = critic(fake).reshape(-1)
```

Since we are using WGAN-GP (Wasserstein Generative Adversarial Networks - Gradient Penalty), we will use Gradient Penalty to adjust the loss. First, we will get the mean of both real and fake image's critique (loss value).

```
# calculate gradient penalty -----
gp = gradient_penalty(critic, real, fake, DEVICE)

# calculate original loss and than apply gradient penalty
original_loss_critic = (
    -(torch.mean(critic_real) - torch.mean(critic_fake))
)
```

Then, we apply the gradient penalty.

```
# apply gradient penalty
loss_critic = original_loss_critic + LAMBDA_GP * gp
```

Explicitly set the gradients to zero before starting to do backpropagation.

```
# set gradients to 0
critic.zero_grad()
loss_critic.backward(retain_graph=True)

# update grads
opt_critic.step()
```

Generative Adversarial Networks: Wasserstein GAN

3. Training Generator

Get the critique value of the fake image (output from the Generator) and calculate the mean of it's loss.



```
# Train gen =====  
output = critic(fake).reshape(-1)  
loss_gen = -torch.mean(output)
```

Explicitly set the gradients to zero before starting to do backpropagation.

```
gen.zero_grad()  
loss_gen.backward()  
opt_gen.step()
```

Github Documentation

<https://github.com/mcandemir/GANs/tree/master/WGAN-GP%20Trainer>


 README.md 

WGAN-GP Trainer

Wasserstein GAN (WGAN), is a type of generative adversarial network that minimizes an approximation of the Earth-Mover's distance (EM) rather than the Jensen-Shannon divergence as in the original GAN formulation.

WGAN-GP is a generative adversarial network that uses the Wasserstein loss formulation *plus* a gradient norm penalty to achieve Lipschitz continuity.

Cat faces



Generative Adversarial Networks: Wasserstein GAN

References

- [1] Tero Karras, Timo Aila, Samuli Laine, Jaakko Lehtinen. *Progressive Growing of GANs for Improved Quality, Stability, and Variation*. 2018.
- [2] Martin Arjovsky, Soumith Chintala, Leon Bottou. *Wasserstein GAN*. 2017.
- [3] Ishaan Gulrajan, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, Aaron Courville. *Improved Training of Wasserstein GANs*. 2017.
- [4] Alec Radford & Luke Metz. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*.