Wasserstein GANs

Suhas Nandiraju and Michael Fatemi

April 2020

1 Introduction

In summary, Wasserstein GAN (WGAN) is a different type of GAN that can generate *more realistic-looking* images than a conventional GAN. WGAN replaces the discriminator with a different loss function, which scores the images by comparing the distribution of images in the generated set with the distribution of images in the input dataset. [1]

2 Brief overview of GANs

"GAN" is short for "Generative Adversarial Network" - a network composed of two neural networks that "compete" to be more accurate than the other. One neural network is the generator, while the other is the discriminator. The generator learns to generate an image that the discriminator thinks is actually real, while the discriminator learns to classify the images as real or fake. In WGAN, the discriminator doesn't learn to classify images as generated or not generated - instead, it learns to classify images based on how *realistic* they look. For this reason, GANs are commonly used to generate data that looks realistic.

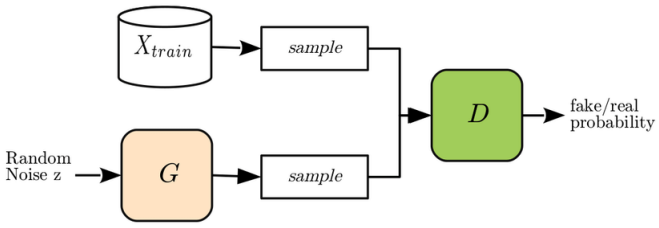

Figure 1: D is the discriminator, and G is the generator.

3 Wasserstein Loss

Basic Overview: The main difference between GAN and WGAN is the use of Wasserstein distance to calculate loss. The Wasserstein distance has been proven through experimentation to be better than standard GAN loss functions to be better suited for image generation in most cases.

For the discriminator: The discriminator works by outputting a number based on the input image: 1 if it believes it's real, and -1 if it believes it's fake. The network tries to maximize its output on real images and minimize its output on fake images. So, the loss is:

```
# Implements Wasserstein loss.
def wasserstein_loss(real_images, fake_images):
    fake_guesses = mean_discriminator_output(fake_images)
    real_guesses = mean_discriminator_output(real_images)
    return real_guesses - fake_guesses
```

For the generator: The generator works by taking random parameters as the input, and generating images based on those parameters. Its loss function is based on how often it can fool the discriminator into believing its images are real.

```
def wasserstein_loss(random_params):
    generated_images = generator_output(random_params)
    return mean_discriminator_output(generated_images)
```

More info can be found at [2].

4 Demo

Example code with Keras can be found at
https://github.com/keras-team/keras-contrib/blob/master/examples/improved_wgan.py. [3]
Example code in FastAI will be below.

5 Conclusion

In conclusion, WGANs are very useful for generating images, like GANs are. The main benefit of using the Wasserstein loss function is that batch normalization is not required. Feel free to use the above FastAI implementation to generate different types of images. Also, if you are interested in a more in depth explanation on how to implement a WGAN, check out the link to the keras code.

```
from fastai.vision import *
from fastai.metrics import error_rate
import os
import numpy as np
from fastai.vision.gan import *

# Load data and process it
# Steps to process the data:
# 1. Crop it to the size
# 2. Organize it into batches
# 3. Normalize the data
def get_data(bs, size, path):
    return GANItemList.from_folder(

        # This is the path
        path,

        # Random parameters used to generate the images
        noise_sz=100)

        # Don't split into training and testing data
        .split_none()

        # Don't label the data (No-Op = no operation)
        .label_from_func(noop)

        # Transform the data
        .transform(
            tfms=[
                # Crop the image to fit the size
                # row_pct = X-axis
                # col_pct = Y-axis
                [crop_pad(size=size, row_pct=(0,1), col_pct=(0,1))]
            ],

            # Image size
            size=size,
            tfm_y=True
        )

        # Organize into batches (BS = batch size)
        .databunch(bs=bs)

        # Normalize the data
        .normalize(
```

```
            stats=[torch.tensor([0.5,0.5,0.5]),
                   torch.tensor([0.5,0.5,0.5])],
            do_x=False, do_y=True
        )

# BS is the batch size.
# Batches are chunks of the original dataset
# that are used during training to even-out outliers.
bs = 64

# Transformations help GANs learn more with less training
# data. Transformations include flipping, dilating,
# and increasing the contrast.
tfms = get_transforms()

# We set a "seed" for the random number generator so it
# generates the same random numbers each time
# we run the code. It helps with consistency.
np.random.seed(7)

# Load the data from the path
data = get_data(bs=128, size=64, path)
data.show_batch(rows=5) #showing some sample data

# Initialize the generator and critic
# in_size: Input size of the image
# n_channels: 3 channels (R, G, B)
generator = basic_generator(in_size=64, n_channels=3, n_extra_layers=1)
critic    = basic_critic   (in_size=64, n_channels=3, n_extra_layers=1)

# This is where the actual GAN comes in
# data: Images to train on
# generator, critic: Networks to use
# switch_eval: False so that the learner stays in training mode
# opt_func: Optimizer. We're using Adam
# wd: Weight decay. Generally 1e-1 but we want to set it to 0 in this case
learn = GANLearner.wgan(
    data,
    generator, critic,
    switch_eval=False,
    opt_func = partial(optim.Adam, betas = (0.,0.99)),
    wd=0
)

# Training the model for 150 epochs with a learning rate of 2e-4
# Epochs are the number of times we pass over the data
```

```
# The learning rate is how quickly we move to converge on
# local minimums for the loss function.
learn.fit(150, 2e-4)

# Generate some sample images
learn.gan_trainer.switch(gen_mode=True)
learn.show_results(ds_type=DatasetType.Train, rows=16, figsize=(8,8))

# Save the model
learn.save("wgan-stage-1")
```

References

[1] James Allingham. Wasserstein gan.

[2] Google. Google machine learning: Gan loss functions.

[3] Keras Team. Demo code in keras.