Evaluation metrics (KID, SWD) and make a comparison study of these metrics for different GAN solutions: Vanilla GAN, WGAN, WGAN-GP

ФИО: **Сюнь Цзини**
Факультет: MEX-MAT *Курс:* Магистр 1 г.о.
Специальность: Математика и компьютерные науки
*Github:*afjzdr/TwoMetricsforGANs

2023/05/15

1 KID and SWD Metrics

1.1 Mathematical Principles of KID and SWD

The Kernel Inception Distance (KID) and the Sliced Wasserstein Distance (SWD) are two metrics used to evaluate the quality of generative models. KID measures the similarity between the generated samples and real data by comparing their feature representations using the Inception network. SWD, on the other hand, quantifies the discrepancy between the generated and real data distributions by computing the Wasserstein distance in a projected space.

The Kernel Inception Distance (KID) measures the similarity between the generated samples G and the real data samples X by comparing their feature representations using the Inception network. It is defined as:

$$KID(G, X) = \|\mu_G - \mu_X\|^2 + Tr(\Sigma_G + \Sigma_X - 2(\Sigma_G\Sigma_X)^{1/2}) \tag{1}$$

where μ_G and μ_X are the mean feature representations of the generated and real data samples respectively, and Σ_G and Σ_X are their respective covariance matrices.

The Sliced Wasserstein Distance (SWD) quantifies the discrepancy between the generated distribution P_G and the real data distribution P_X by computing the Wasserstein distance in a projected space. It is defined as:

$$SWD(P_G, P_X) = w \sim \mathcal{W}E\left[\left\|\int_R \phi_w(t)dP_G(t) - \int_R \phi_w(t)dP_X(t)\right\|\right] \tag{2}$$

where $\mathcal{W}$ represents the set of all projection matrices, $\phi_w(t)$ is a set of random Fourier features, and $dP_G(t)$ and $dP_X(t)$ are the projected measures of the generated and real data distributions respectively.

1.2 Using KID and SWD for Model Evaluation

To assess the quality of a generative model, KID and SWD can be computed on a set of generated samples and compared against the corresponding metrics calculated on real data. Lower values of KID and SWD indicate a better model performance, as it suggests that the generated samples are closer to the real data distribution.

To assess the quality of a generative model, KID and SWD can be calculated on a set of generated samples and compared with the corresponding metrics calculated on real data. Lower KID and SWD values indicate better model performance, as it indicates that the generated samples are closer to the distribution of the real data. In this experiment, the three models vanilla GAN, WGAN, and WGAN-GP will be trained separately using the training set of CIFAR-10, and after every 10 epochs, the generated set generated by the model with the same specifications as the test set of CIFAR-10 will be computed and compared with the test set to calculate the KID and SWD, and thus observed.

1.3 Differences Between KID/SWD and IS/FID

KID and SWD offer some advantages over other metrics like Inception Score (IS) and Fréchet Inception Distance (FID). KID and SWD do not rely on a pretrained classifier or feature extractor, making them more robust to model biases. Additionally, KID and SWD consider both local and global statistics, allowing for a more comprehensive evaluation of the generated samples.

1.4 Code KID/SWD

In this section, we provide the Python code implementations for calculating the Kernel Inception Distance (KID) and the Sliced Wasserstein Distance (SWD) metrics.

```
class KID:
    def __init__(self, real_dataset, generated_dataset, batch_size=64):
        self.real_dataset = real_dataset
        self.generated_dataset = generated_dataset
        self.batch_size = batch_size

    def compute_features(self, dataset):
        # Computes a feature function that is used to compute a feature
            representation for a given dataset. This function uses the pre-trained
             DenseNet-121 model (models.densenet121(pretrained=True)) as the
            feature extractor. It traverses batches of the dataset, passes the
            input data to the model, obtains the output features of the model and
            stores these features in a list. Finally, by stitching these features
            over dimension 0, they are converted to a tensor and returned.
    def compute_kid(self):
        # The compute KID metric function, which calculates the Kernel Inception
            Distance (KID) between the generated dataset and the real dataset. It
            calls the compute_features function to calculate the feature
            representations of the true and generated datasets respectively. It
            then calculates the mean and covariance of the features of the true
            and generated datasets and calculates the mean difference and
            covariance difference. Finally, it calculates the value of the KID
            metric by computing the van, trace and square root terms in the KID
            formula and returns it as a floating point number.
        return kid.item()

class SWD:
    def __init__(self, real_dataset, generated_dataset, batch_size=64, num_slices
        =10):
        self.real_dataset = real_dataset
        self.generated_dataset = generated_dataset
        self.batch_size = batch_size
        self.num_slices = num_slices

    def compute_features(self, dataset):
        # Computes a feature function that is used to compute a feature
            representation for a given dataset. This function uses the pre-trained
             DenseNet-121 model (models.densenet121(pretrained=True)) as the
            feature extractor. It traverses batches of the dataset, passes the
            input data to the model, obtains the output features of the model and
            stores these features in a list. Finally, by stitching these features
            over dimension 0, they are converted to a tensor and returned.
    def compute_swd(self):
        # The compute SWD metric function for calculating the Sliced Wasserstein
            Distance (SWD) between the generated dataset and the true dataset. It
```

```
        uses the compute_features function to calculate feature
        representations for the real and generated datasets respectively. It
        then slices the features into multiple slices and calls the
        _compute_wasserstein_distance function to calculate the Wasserstein
        distance between the slices. Finally, the Wasserstein distance is
        summed to obtain the value of the SWD metric and returned as a
        floating point number.
    def _slice_data(self, data):
        # ~~~
        return data_slices

    def _compute_wasserstein_distance(self, real_slices, generated_slices):
        # ~~~
        return wasserstein_distance

    def _pairwise_distance(self, x, y):
        # ~~~
        return distance_matrix
```

2 Comparison of Vanilla GAN, WGAN, and WGAN-GP Architectures

In this section, we analyze and compare the structures of three popular generative models: Vanilla GAN, Wasserstein GAN (WGAN), and Wasserstein GAN with Gradient Penalty (WGAN-GP). Each of these models employs different techniques to improve training stability and address the mode collapse issue commonly found in GANs.

2.1 Vanilla GAN

Vanilla GAN is the original formulation proposed by Goodfellow et al. (2014). It consists of two main components: a generator network and a discriminator network. The generator aims to generate realistic samples from random noise, while the discriminator tries to distinguish between the generated samples and real data. The training objective is a minimax game where the generator tries to maximize the discriminator's error while the discriminator tries to minimize its error.

However, Vanilla GANs often suffer from training instability and mode collapse. The discriminator can become too strong, overpowering the generator and resulting in limited diversity in the generated samples.

2.2 Wasserstein GAN (WGAN)

Wasserstein GAN, proposed by Arjovsky et al. (2017), addresses the instability and mode collapse issues of Vanilla GANs by introducing the Wasserstein distance as a measure of discrepancy between the generated and real data distributions. Instead of training the discriminator to output probabilities, WGAN trains the discriminator to output a real-valued score, representing the estimated Wasserstein distance.

To enforce the Lipschitz constraint necessary for the Wasserstein distance, weight clipping is often applied to the discriminator's parameters in WGAN. However, this approach can lead to suboptimal performance and difficulties in finding an appropriate clipping value.

2.3 Wasserstein GAN with Gradient Penalty (WGAN-GP)

Wasserstein GAN with Gradient Penalty, proposed by Gulrajani et al. (2017), further improves upon WGAN by replacing weight clipping with a gradient penalty regularization term. The gradient penalty encourages smoothness in the discriminator's output and avoids the need for manually setting a clipping value.

The gradient penalty term penalizes the norm of the discriminator's gradients with respect to interpolated points between real and generated samples. This encourages the discriminator to have a more meaningful gradient throughout the input space, leading to improved training stability and better sample quality.

WGAN-GP has been shown to achieve more stable training compared to both Vanilla GAN and WGAN, while also generating higher-quality samples with increased diversity.

3 Model Quality Evaluation on CIFAR-10 Dataset

We trained Vanilla GAN, WGAN, and WGAN-GP models on the CIFAR-10 dataset. The objective is to assess the quality of each model using KID and SWD metrics.

3.1 Experimental Setup

3.1.1 Vanilla GAN

The generator in this GAN model consists of several fully connected layers. It takes a random noise vector as input and progressively increases its dimensions through the hidden layers. Each hidden layer is followed by a leaky ReLU activation function, which helps introduce non-linearity. The final layer of the generator is a linear layer followed by a hyperbolic tangent (tanh) activation function. This produces the generated fake images as the output. The output is reshaped to match the shape of CIFAR-10 images (-1, 3, 32, 32).

On the other hand, the discriminator takes an image tensor as input and aims to classify whether the input image is real or fake. It consists of fully connected layers that gradually decrease the dimensions from the image dimension to a single output representing the probability of the input being real. Each hidden layer is followed by a leaky ReLU activation function. The output layer of the discriminator uses a sigmoid activation function to produce a probability value between 0 and 1.

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim, hidden_dim * 2),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim * 2, hidden_dim * 4),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim * 4, image_dim),
            nn.Tanh(),
        )

    def forward(self, x):
        return self.main(x).view(-1, 3, 32, 32)

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(image_dim, hidden_dim * 4),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim * 4, hidden_dim * 2),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        return self.main(x.view(-1, image_dim))
```

3.1.2 WGAN

The generator takes a 100-dimensional random noise vector as input and outputs a 3x32x32 image. The discriminator takes a 3x32x32 image as input and outputs a scalar value indicating how real or fake the image is.

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.fc = nn.Linear(100, 512*2*2)
        self.deconv1 = nn.ConvTranspose2d(512, 256, 4, 2, 1)
        self.deconv2 = nn.ConvTranspose2d(256, 128, 4, 2, 1)
        self.deconv3 = nn.ConvTranspose2d(128, 64, 4, 2, 1)
        self.deconv4 = nn.ConvTranspose2d(64, 3, 4, 2, 1)
        self.bn1 = nn.BatchNorm2d(256)
        self.bn2 = nn.BatchNorm2d(128)
        self.bn3 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, z):
        z = self.fc(z)
        z = z.view(-1, 512, 2 ,2)
        z = self.relu(self.bn1(self.deconv1(z)))
        z = self.relu(self.bn2(self.deconv2(z)))
        z = self.relu(self.bn3(self.deconv3(z)))
        z = self.tanh(self.deconv4(z))
        return z

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, 4, 2, 1)
        self.conv2 = nn.Conv2d(64, 128, 4, 2, 1)
        self.conv3 = nn.Conv2d(128, 256, 4, 2, 1)
        self.conv4 = nn.Conv2d(256, 512, 4, 2, 1)
        self.fc = nn.Linear(512*2*2, 1)
        self.lrelu = nn.LeakyReLU(0.2)

    def forward(self, x):
        x = self.lrelu(self.conv1(x))
        x = self.lrelu(self.conv2(x))
        x = self.lrelu(self.conv3(x))
        x = self.lrelu(self.conv4(x))
        x = x.view(-1, 512*2*2)
        x = self.fc(x)
        return x
```

3.1.3 WGAN-GP

Generator: The generator consists of a fully connected layer followed by several transposed convolutional layers with batch normalization and ReLU activation functions. It takes a random noise vector as input and aims to generate realistic images.

Discriminator: The discriminator consists of several convolutional layers with leaky ReLU activation functions. It takes an image tensor as input and aims to classify whether the input image is real or fake.

3.1.4 Key differences in models

The main structural differences between vanilla GAN, WGAN, and WGAN-GP lie in the discriminator. In vanilla GANs, the discriminator is a traditional binary classifier. In WGANs, the discriminator outputs the Wasserstein distance. WGAN-GPs add gradient penalty calculations to the WGAN framework to enforce Lipschitz continuity. The generator structures in these GAN variants are generally similar, focusing on transforming noise into realistic samples.

3.2 Results and Analysis

3.2.1 Overview of results

We trained vanilla GAN, WGAN, and WGAN-GP for 100 epochs, respectively, and tallied the final obtained models and generated data metrics as follows:

Table 1: KID, SWD

CIFAR-10	vanilla GAN	WGAN	WGAN-GP
Epochs	100	100	100
g_{loss}	1.4116	-9.1989	-2.5692
d_{loss}	1.1926	-2.8056	-2.3693
KID	-229002.6562	-195936.5312	-152039.1562
SWD	0.4740	0.3664	0.2310

where g_{loss} and d_{loss} are the loss of generators and discriminators.
Five images were then randomly selected from the datasets generated by each model for display:

Figure 1: vanilla GAN

Figure 2: WGAN

Figure 3: WGAN-GP

3.2.2 Visualisation of the training process

The KID and SWD indicators calculated during the training of the three models are visualised below:

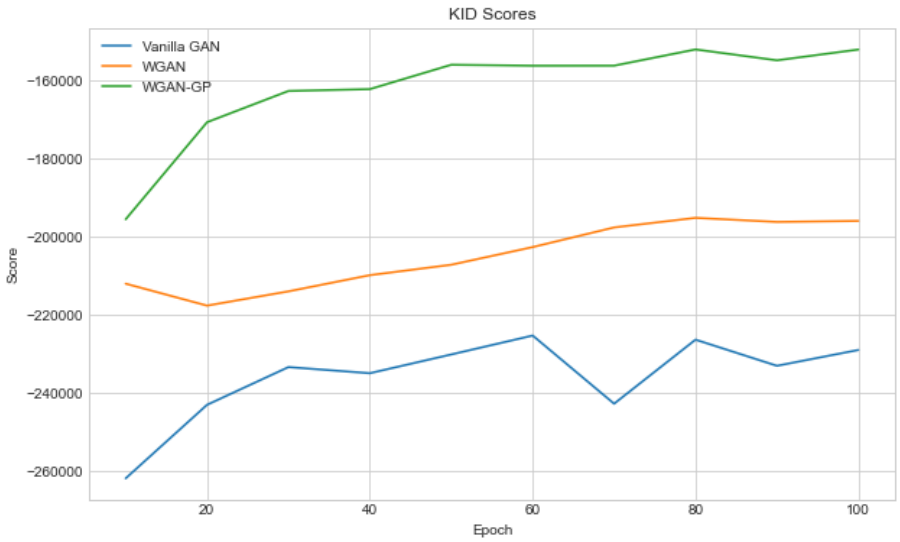

Figure 4: kid scores

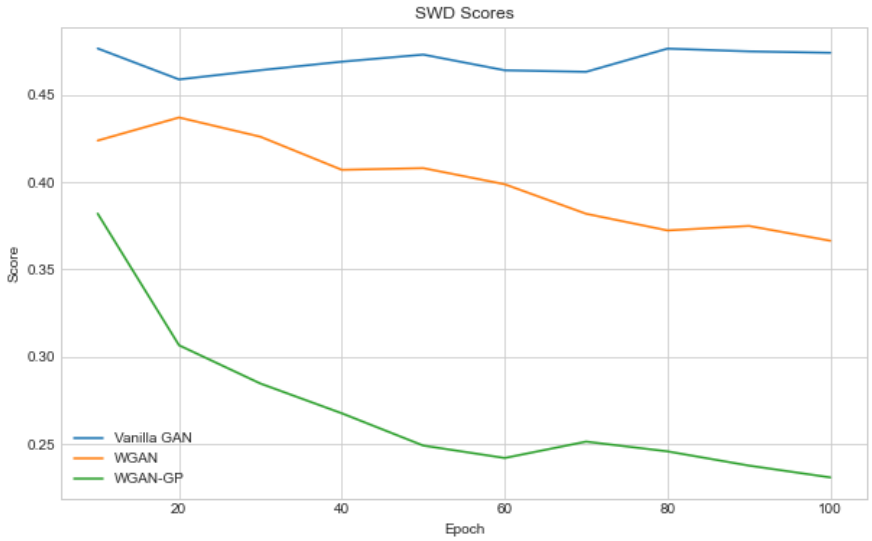

Figure 5: swd scores

3.2.3 Analysis

Both the KID and SWD metrics are close to 0, which proves that the quality of the model is better. Therefore, it can be seen from these two metrics during the training process that the quality of the generated set of vanilla GAN, WGAN and WGAN-GP models is getting better over 100 epochs, and the performance of WGAN-GP is the best.

1. vanilla GAN: The performance of the vanilla GAN model is the worst, with a poor trend in both metrics, presumably due to a mode collapse problem.

2. WGAN: The overall trend of the WGAN model is getting better, but the change is not significant and also reflects the fact that the model does not train with good performance within 100 epochs.

3. WGAN-GP: The WGAN-GP model has the most pronounced trend for the two metric transformations, and as can also be seen by the final visual measure of the generated samples, the results are much better than the first two models.

In general, the two GANs model metrics, KID and SWD, give a good indication of the quality of the datasets generated by the model at each stage, and thus reflect the quality of the model at this time, but for models with older technology, such as vanilla GAN, which may suffer from mode collapse (mode collapse) problems, it can be seen that the model at this stage is not of high quality, but does not reflect exactly what kind of problems are occurring with the model at this time. In addition, both KID and SWD metrics require the import of a picture feature extraction network (DenseNet -121 was used in this experiment), which requires a certain amount of GPU resources and slows down the overall training speed of the model.

Contents