Fast Fréchet Inception Distance

Alexander Mathiasen* **Frederik Hvilshøj***

Abstract

The Fréchet Inception Distance (FID) has been used to evaluate thousands of generative models. We present a novel algorithm, FastFID, which allows fast computation and backpropagation for FID. FastFID can efficiently (1) evaluate generative model *during* training and (2) construct adversarial examples for FID.

1 Introduction

Generative modeling is a popular approach to unsupervised learning, with applications in, e.g., computer vision [21] and drug discovery [19]. A key difficulty for generative models is to evaluate their performance, i.e., how good is the generative model approximating the data distribution? For image data, it is most common to use the Frechet Inception Distance (FID) [14]. FID is computed using m "fake" samples from the model distribution and n "real" samples from the data distribution.

FID is used in two ways [14]. Most commonly, FID is used to compare generative models *after* training. However, FID can also be used to monitor improvements of a single generative model *during* training. Monitoring improvements *during* training is particularly useful when training Generative Adversarial Networks [12], which normally lack a numeric indicator for whether the generators performance improves. For model comparison *after* training one typically use 50000 samples, which takes a long time. For monitoring improvement *during* training, we find that it is sufficient to use surprisingly few fake samples m if the number of real samples n remains large.

Decreasing m appropriately reduces computation time from approximately $60s$ to $10s$. We further reduce computation time to approximately $0.2s$ by introducing a novel algorithm FastFID. FastFID supports gradient computations which allow efficient construction of adversarial examples for FID, see Figure 1. FastFID generalizes to similar distances like Fréchet ChemNet Distance (FCD) [20].

(a) Adversarial Noise (b) Adversarial Example

Figure 1: First row is original noise (bad FID) and original data (good FID). Second row contains modified noise (good FID) and modifid data (bad FID). See Section 5 for details.

Preprint.

*Aarhus University, {alexander.mathiasen, fhvilshoj}@gmail.com

2 Fast Fréchet Inception Distance

The FID between the model distribution P_{model} and the real data distribution P_{data} is computed as follows. Draw "fake" model samples $f_1, ..., f_m \sim P_{model}$ and "real" data samples $r_1, ..., r_n \sim P_{data}$. Encode all samples f_i and r_i by computing activations $A(f_i)$ and $A(r_i)$ of the final layer of the pretrained Inception network [23]. Compute the sample means μ_1, μ_2 and the sample covariance matrices Σ_1, Σ_2 of the activations $A(f_i), A(r_i)$. The FID is then the Wasserstein distance [11] between the two multivariate normal distributions $N(\mu_1, \Sigma_1)$ and $N(\mu_2, \Sigma_2)$.

$$W(N(\mu_1, \Sigma_1), N(\mu_2, \Sigma_2))^2 = ||\mu_1 - \mu_2||_2^2 + \text{tr}(\Sigma_1) + \text{tr}(\Sigma_2) - 2 \cdot \text{tr}(\sqrt{\Sigma_1 \Sigma_2}) \tag{1}$$

For evaluation during training, the original implementation use 10000 samples by default [14]. On our workstation,[2] this cause the FID evaluation to take approximately $20s$. Of the $20s$, it takes approximately $10s$ to compute Inception encodings and approximately $10s$ to compute $\text{tr}(\sqrt{\Sigma_1 \Sigma_2})$.

The real data samples $r_1, ..., r_n$ does not change *during* training, so we only need to compute their Inception encodings once. The $10s$ spent computing the Inception encodings is only the time it takes to encode the fake model samples $f_1, ..., f_m$. It is thus sufficient to reduce the number of fake samples m to reduce the time spent computing Inception encodings. For example, if we reduce m from 10000 to 128 we reduce the time spent computing Inception encodings from $10s$ to $0.1s$. However, it still takes around $10s$ to compute $\text{tr}(\sqrt{\Sigma_1 \Sigma_2})$. FastFID mitigates this issue by efficiently computing $\text{tr}(\sqrt{\Sigma_1 \Sigma_2})$ without explicitly computing $\sqrt{\Sigma_1 \Sigma_2}$. Section 2.1 outlines how previous work computed $\sqrt{\Sigma_1 \Sigma_2}$ and Section 2.2 introduce our novel algorithm that efficiently computes $\text{tr}(\sqrt{\Sigma_1 \Sigma_2})$.

2.1 Previous algorithm

The original FID implementation [14] computes $\text{tr}(\sqrt{\Sigma_1 \Sigma_2})$ by explicitly constructing $\sqrt{\Sigma_1 \Sigma_2}$. The matrix square root is computed using SCIPY.LINALG.SQRTM [24] which implements an extension of the algorithm from [3] which is rich in matrix-matrix operations [10]. The algorithm starts by computing the following Schur decomposition.

$$\Sigma_1 \Sigma_2 = QVQ^T \text{ where } Q^T = Q^{-1} \text{and } V \text{ is triangular.} \tag{2}$$

The algorithm then computes a triangular matrix U such that $U^2 = V$ by exploiting the triangular structure of both U and V. In particular, the triangular structure implies the following triangular equations $U_{ii}^2 = V_{ii}$ and $U_{ii}U_{ij} + U_{ij}U_{jj} = V_{ij} - \sum_{k=i+1}^{j-1} U_{ik}U_{kj}$ [10]. The equations can be solved wrt. U one superdiagonal at a time. The resulting U yields a matrix square root of the initial matrix.

$$\sqrt{\Sigma_1 \Sigma_2} = QUQ^T \tag{3}$$

Time Complexity. Computing the Schur decomposition of a $d \times d$ matrix $\Sigma_1 \Sigma_2$ takes $O(d^3)$ time. The resulting triangular equations can then be solved wrt. U in $O(d^3)$ time. The entire matrix square root computation then takes $O(d^3)$ time. The size of d depends on the network that encodes the data. For example, FID uses the Inception network which has $d = 2048$ while FCD uses the ChemNet network which has $d = 512$. On our workstation, the different values of d cause the square root computations to take approximately $10s$ for FID and $1s$ for FCD.

Uniqueness. The matrix square root $\sqrt{M}$ is defined to be any matrix that satisfies $\sqrt{M}\sqrt{M} = M$. The square root of a matrix is in general not unique, i.e., some matrices have many square roots. The above algorithm does not necessarily find the same square root matrix if it is run several times, because the Schur decomposition is not unique. Furthermore, when computing $U_{ii} = \sqrt{T_{ii}}$ one has the freedom to choose both $\pm|\sqrt{T_{ii}}|$. The SCIPY.LINALG.SQRTM implementation choose $U_{ii} = |\sqrt{T_{ii}}|$. Our implementation of the algorithm we present in Section 2.2 computes the trace $\text{tr}(\sqrt{\Sigma_1 \Sigma_2})$ such that it agrees with SCIPY.LINALG.SQRTM up to numerical errors.

[2] RTX 2080 Ti with Intel Xeon Silver 4214 CPU @ 2.20GHz, computing FID on CelebA [16] images using the following implementation of FID `https://github.com/hukkelas/pytorch-frechet-inception-distance` with precomputed Inception encodings of the real data.

2.2 Our algorithm

This subsection presents an algorithm that computes $\text{tr}(\sqrt{\Sigma_1\Sigma_2})$ fast. The high-level idea: construct a "small" matrix M such that the eigenvalues $\lambda_i(M)$ satisfy $\sum_i |\sqrt{\lambda_i(M)}| = \text{tr}(\sqrt{\Sigma_1\Sigma_2})$. Since M is small, we can compute its eigenvalues faster than we can compute the matrix square root $\sqrt{\Sigma_1\Sigma_2}$.

Let $X_1 \in \mathbb{R}^{d\times m}$ be a matrix with columns $A(f_1), ..., A(f_m)$, let $X_2 \in \mathbb{R}^{d\times n}$ be a matrix with columns $A(r_1), ..., A(r_n)$ and let $\mathbb{1}_k$ be the $1 \times k$ all ones vector. The sample covariance matrices Σ_1 and Σ_2 can then be computed as follows:

$$\Sigma_i = C_i C_i^T, \quad \text{where} \quad C_1 = \frac{1}{\sqrt{m-1}}(X_1 - \mu_1 \mathbb{1}_m) \quad \text{and} \quad C_2 = \frac{1}{\sqrt{n-1}}(X_2 - \mu_2 \mathbb{1}_n) \tag{4}$$

This allows us to write $\Sigma_1\Sigma_2 = C_1C_1^TC_2C_2^T$. Recall that the eigenvalues of AB are equal to the eigenvalues of BA if both AB and BA are square matrices [17]. The eigenvalues of the $d \times d$ matrix $C_1C_1^TC_2C_2^T$ are thus the same as the eigenvalues of the $m \times m$ matrix $M = C_1^TC_2C_2^TC_1$. The matrix M is small in the sense that we will use $m \ll d$, for example, for FID we often use $m = 128$ fake samples while $d = 2048$.

We now show that $\sum_i |\sqrt{\lambda_i(M)}| = \text{tr}(\sqrt{\Sigma_1\Sigma_2})$ if $\sqrt{\Sigma_1\Sigma_2}$ is computed by SCIPY.LINALG.SQRTM. Since $\text{tr}(\sqrt{\Sigma_1\Sigma_2}) = \sum_i \lambda_i(\sqrt{\Sigma_1\Sigma_2})$ it is sufficient to show that the eigenvalues of $\sqrt{\Sigma_1\Sigma_2}$ are equal to the positive square root of the eigenvalues of $\Sigma_1\Sigma_2$, that is, $\lambda_i(\sqrt{\Sigma_1\Sigma_2}) = |\sqrt{\lambda_i(\Sigma_1\Sigma_2)}|$.

Recall that $\sqrt{\Sigma_1\Sigma_2} = QUQ^T$ where $\Sigma_1\Sigma_2 = QVQ^T$, $U^2 = V$ and $U_{ii}^2 = V_{ii}$. Since both U and V are triangular they have their eigenvalues on the diagonal, which means that $\lambda_i(\sqrt{\Sigma_1\Sigma_2}) = U_{ii} = \sqrt{V_{ii}} = \sqrt{\lambda_i(\Sigma_1\Sigma_2)}$. Recall that SCIPY.LINALG.SQRTM choose $U_{ii} = |\sqrt{V_{ii}}|$ and we get $\lambda_i(\sqrt{\Sigma_1\Sigma_2}) = |\sqrt{\lambda_i(\Sigma_1\Sigma_2)}|$ as wanted. Note that even though the Schur decomposition $\Sigma_1\Sigma_2 = QUQ^T$ is not unique, U will always have the eigenvalues of $\Sigma_1\Sigma_2$ on its diagonal and thus preserve the trace. Putting everything together, we realize that $\text{tr}(\sqrt{\Sigma_1\Sigma_2}) = \sum_{i=1}^{m-1} |\sqrt{\lambda_i(C_1^TC_2C_2^TC_1)}|$.

For completeness, we provide pseudo-code in Algorithm 1. The algorithm can be modified to compute FCD by simply changing the Inception network to the ChemNet network.

Time Complexity. Computing $M = (C_1^TC_2)(C_2^TC_1)$ takes $O(mdn+m^2n)$ time. The eigenvalues of M can be computed in $O(m^3)$ time, giving a total time complexity of $O(mdn + m^2n + m^3)$. If we use a large number of real samples $n \gg d$, we can precompute $\Sigma_2 = C_2C_2^T$ and compute $M = C_1^T\Sigma_2C_1$ in $O(d^2m)$ time, giving a total time complexity of $O(d^2m + m^3)$.

Algorithm 1 Fast Fréchet Inception Distance

Input: $f_1, ..., f_m \sim P_{model}$, the Inception network Net(x) and precomputed μ_2, C_2.

// Compute network activations

Compute $A(f_i) = \text{Net}(f_i)$ and let X_1 be a matrix with columns $A(f_1), ..., A(f_m)$.

// Compute mean.

$\mu_1 = \frac{1}{m}\sum_{i=1}^{m}(X_1)_i$.

// Compute trace of square root matrix.

$C_1 = \frac{1}{\sqrt{m-1}}(X_i - \mu_i\mathbb{1})$ as in Equation (4).

$S = \text{library.eigenvalues}((C_1^TC_2)(C_2^TC_1))$

$\text{tr}(\sqrt{\Sigma_1\Sigma_2}) = \sum_{i=1}^{m-1} |\sqrt{S_i}|$

// Compute trace of both covariance matrices.

$\text{tr}(\Sigma_1) = \sum_{i=1}^{m} \text{row}_i(C_1)^T\text{row}_i(C_1)$

$\text{tr}(\Sigma_2) = \sum_{i=1}^{n} \text{row}_i(C_2)^T\text{row}_i(C_2)$

return $||\mu_1 - \mu_2||_2^2 + \text{tr}(\Sigma_1) + \text{tr}(\Sigma_2) - 2 \cdot \text{tr}(\sqrt{\Sigma_1\Sigma_2})$

Computing Gradients. If the network Net(x) used in Algorithm 1 supports gradients with respect to its input, as the Inception network does, it is possible to compute gradients with respect to the input samples f_i. If f_i were constructed by a generative model, one can even compute gradients wrt. the parameters of the generative model. If Algorithm 1 is implemented with autograd support, e.g., through PyTorch [18] or TensorFlow [1], these gradients are computed automatically. For example, we implemented Algorithm 1 in PyTorch and used autograd to compute gradients when constructing adversarial examples in Section 5.

Potential Further Speed-ups. In this section, we have focused on decreasing the time consumption of $\text{tr}(\sqrt{\Sigma_1\Sigma_2})$. As a result, the majority of the time consumption of Algorithm 1 is spent computing Inception encodings. The Inception encodings could be further speed up by, e.g., compressing [6] the used network at the cost of introducing some small error.

3 Experimental Performance of FastFID

3.1 Speed-Up

In Section 2, we presented a fast algorithm to compute $\text{tr}(\sqrt{\Sigma_1\Sigma_2})$, which can be used to speed up the computation of both FID and FCD. In this subsection, we compare the running time of our PyTorch [18] implementation of Algorithm 1 against open-source implementations.[3] The algorithms are compared as the number of different fake samples varies $m = 256, 128, ..., 8$.

We used images from the CelebA dataset [15] when timing FID. We used SMILES molecules from the MOSES dataset [19] when timing FCD. For both FID and FCD, we precomputed Σ_2 with 10000 real samples. We also timed $\text{tr}(\sqrt{C_1^T\Sigma_2 C_1})$ in isolation, by using $\Sigma_2 \in \mathbb{R}^{d\times d}$ with $N(0,1)$ entries and computed C_1 from Equation (4) where $X_1 \in \mathbb{R}^{d\times n}$ has $N(0,1)$ entries. To make timings of $\text{tr}(\sqrt{C_1^T\Sigma_2 C_1})$ comparable with FID, we chose $d = 2048$ so it matches the Inception network.

For each m, we ran one warmup round and then repeated the experiment 100 times. Figure 2 plots average running time with standard deviation as error bars.[4] The previous algorithm takes roughly the same amount of time for all m. This is expected since the matrix square root takes up most of the computation time, and the computation time does not depend on m. Our algorithm takes less time as m decrease. This is expected since our algorithm takes $O(m^3 + d^2m)$ time for precomputed Σ_2.

For all m, our algorithm is at least 25 times faster than the previous algorithms, and in some cases up to hundreds or even thousands of times faster. We emphasize that both algorithms were given the same task and computed the same thing, our algorithm was just faster.

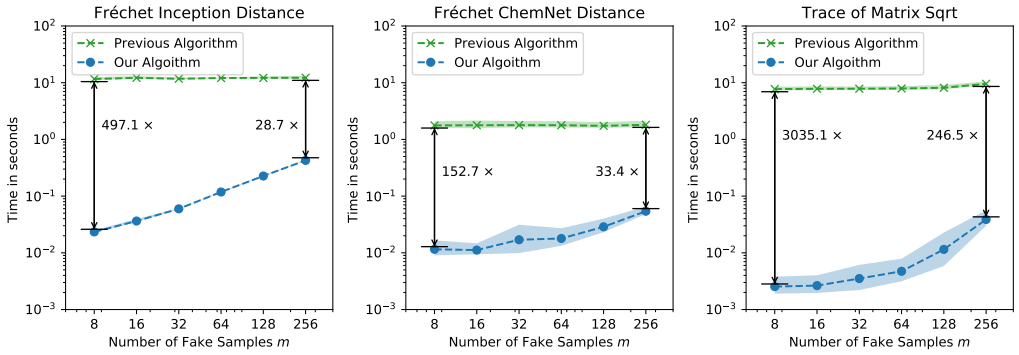


Figure 2: Comparison of SCIPY.LINALG.SQRTM and Algorithm 1 for different batch sizes. Left: Time it takes to compute Fréchet Inception Distance. Middle: Time it takes to compute Fréchet ChemNet Distance. Right: Time it takes to compute trace of matrix square root.

[3] See SCIPY.LINALG.SQRTM from [24], `https://github.com/insilicomedicine/fcd_torch` and `https://github.com/hukkelas/pytorch-frechet-inception-distance`

[4] For our algorithm, we plot $[\mu - \sigma/2, \mu + \sigma]$ instead of $\mu \pm \sigma$ to avoid clutter caused by the logarithmic scaling. This means that the real running time of our algorithm is sometimes a little bit better than visualized.

3.2 Numerical Error

To investigate the numerical error of Algorithm 1 and SCIPY.LINALG.SQRTM, we ran an experiment where the square root can easily be computed. If we choose $C_1 = C_2$ then $C_1 C_1^T C_2 C_2^T = (C_1 C_1^T)^2$ is positive semi-definite and has an unique positive semi-definite square root $\sqrt{(C_1 C_1^T)^2} = C_1 C_1^T$. Furthermore, it is exactly this positive semi-definite square root SCIPY.LINALG.SQRTM computes, since the implementation chooses the positive eigenvalues $U_{ii} = |\sqrt{V_{ii}}|$ (see Section 2.1 for details). We can then investigate the numerical errors by comparing the ground truth $\text{tr}(C_1 C_1^T)$ with the result from both Algorithm 1 and SCIPY.LINALG.SQRTM. To amplify the numerical errors we used 32-bit floating points instead of 64-bit floating points.

The experiment was repeated for $m = 32, 64, 128, 256$ number of fake samples, were C_1 was computed as done in Section 3.1. We report the ground truth $\text{tr}(C_1 C_1^T)$ and the absolute numerical error caused by Algorithm 1 and SCIPY.LINALG.SQRTM, e.g., we report $|\text{tr}(C_1 C_1^T) - \text{tr}(\text{SCIPY.LINALG.SQRTM}(C_1 C_1^T C_1 C_1^T))|$. See results in Table 1. The numerical error of Algorithm 1 is at least 1000 times smaller than that of SCIPY.LINALG.SQRTM. We suspect Algorithm 1 is more numerically stable because it computes eigenvalues of a "small" $m \times m$ matrix instead of computing a Schur decomposition of the "full" $d \times d$ matrix.

Table 1: Numerical error of SCIPY.LINALG.SQRTM and Algorithm 1 for different number of fake samples m. We also report the ground truth so the percentage wise numerical error can be inferred.

m	Ground Truth	Error of SCIPY.LINALG.SQRTM	Error of Algorithm 1
32	62955.6250	300.0112	0.0000
64	128947.5625	408.3399	0.0078
128	259586.1250	565.4011	0.0156
256	523360.3438	785.0262	0.0312

4 Efficiently Evaluating Generative Models During Training

4.1 Fréchet Inception Distance

It can be hard to determine whether a generative model improves during training. This is particularly difficult for GANs, where one does not have a single numeric value that measures error. For example, consider left plot in Figure 3, which displays the error of discriminator/generator during training of a DCGAN [21] on CelebA [16] at 64×64 resolution. This looks quite different to supervised error functions like mean squared error, which typically drops steadily during training. Both the discriminator error and the generator error has no clear interpretation. This makes it difficult to determine whether training improves the performance of the generator.

To mitigate this issue, [14] used FID to evaluate a DCGAN [21] during training. We similarly computed FID every 10 mini-batch updates using $m = 10000$, see the right plot in Figure 3.[5] The FID drops steadily during training, which indicates that training does indeed improve the performance of the generator, at least as measured by FID. However, computing FID with $m = 10000$ is computationally expensive. Each FID computation took at least $20s$, while a single mini-batch update of the DCGAN took $0.09s$. This means the FID computations took at least 20 *times* the training time, even though we only computed FID once every 10 iterations. To speed up the evaluation, we computed FID with Algorithm 1 using just $m = 128$ fake samples, while still using the $n = 10000$ precomputed activations on the validation set. The resulting FID computation took $0.2s$, at least 100 times faster. As a result, the FID computations took no more than 0.25 *times* the training time.

Let FID_m denote FID evaluated with m fake samples. We plot both FID_{128} and FID_{10^4} during training of the DCGAN in Figure 4. The left plot shows both FIDs during the first epoch (1500 mini-batch updates). Both FIDs drop during training, and they qualitatively appear to be very similar. The middle plot shows that both FID losses behave similarly throughout training.

[5] The source-code from [14] use $m = 10000$ by default, see `https://github.com/bioinf-jku/TTUR/blob/d4baae8c095876c01c032c3c53d5542b69887a9b/DCGAN_FID_batched/model.py#L32`. However, [14] report training FID since they precompute activations on the entire training set. To take overfitting into account, we use a validation set of size $n = 10000$ and precompute activations on the validation set.

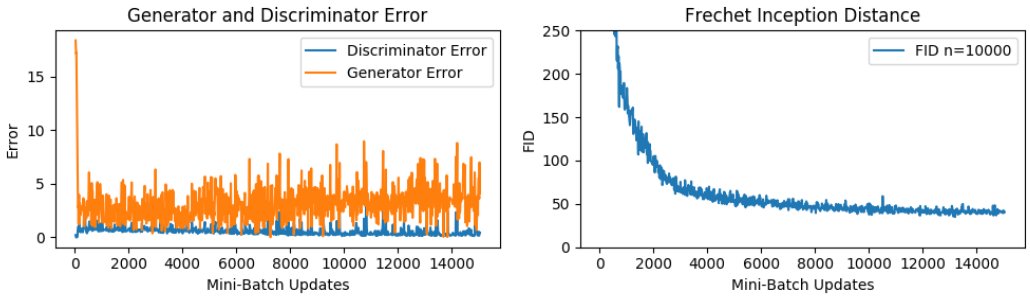


Figure 3: Left: discriminator and generator error when training DCGAN [21] on CelebA [16]. Based on the error curves, it is difficult to determine whether (and how much) the generator is improving. Right: FID during training of DCGAN, FID consistently drops which indicates training does improve the generator.

To quantify the error we get by using FID_{128} instead of FID_{10^4}, we attempt to measure how good FID_{128} is at predicting FID_{10^4}. For simplicity, we fitted a linear regression model $f(FID_{128}) = a{\cdot}FID_{128}{+}b$ to approximate FID_{10^4}. The linear regression model solves to $f(x) = 1.086x{-}54.165$ which attains correlation coefficient $r = 0.995$ and a mean squared error of 0.0032. The right plot in Figure 4 shows FID_{128} on x-axis and FID_{10^4} on y-axis and the linear regression model.

The linear model suggests that the computed FID_{10^4} and FID_{128} are roughly related by an offset. It thus seems reasonable to use FID_{128} as a measure of relative generative improvement during training. We emphasize that the $100\times$ speed-up typically makes the difference as to whether such quantitative evaluation is practically feasible or not. One of the main difficulties when debugging and experimenting with GANs, is the lack of a single numeric measure of generative performance that can be frequently evaluated at a low cost, this is now possible with FID_{128} using Algorithm 1.

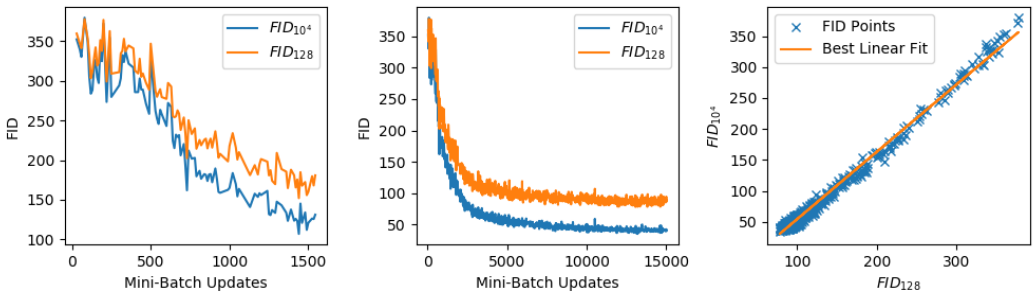


Figure 4: Comparison of FID computed with $m = 128$ and $m = 10^4$ when evaluating DCGAN [21] trained on CelebA [16] at resolution 64x64. Left: First 1600 mini-batch updates, most peaks co-occur. Middle: Training throughout 10 epochs. Right: Relationship between FID evaluated with $m = 128$ and $m = 10^4$ fake samples, and the best linear fit $r = 0.995$.

4.2 Frechet ChémNet Distance

On image data it is often possible for humans to qualitatively evaluate whether model samples improve during the early stages of training. However, it seems harder for humans to carry out such qualitative assessment when the generated samples are, e.g., molecules. This motivates the need for quantitative evaluation of molecules during training of generative models. In this section, we demonstrate that Algorithm 1 also allows efficient computation of FCD. For simplicity, we retrained the Variational AutoEncoder from [19] on the MOSES dataset and compared FCD_{128} against FCD_{10^4}. The results are visualized in Figure 5 following the same structure as Figure 4. It took around $1.87s$ to compute FCD_{10^4} and only $0.055s$ to compute FCD_{128}, at least 30 times faster.

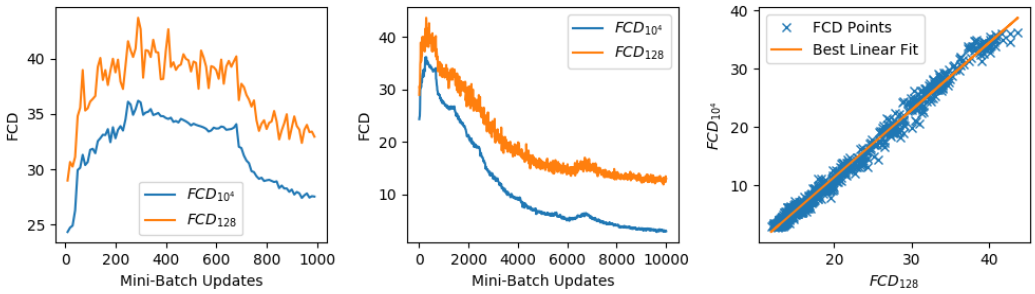


Figure 5: Comparison of FCD computed with $m = 128$ and $m = 10^4$ when evaluating VAE trained on MOSES [19]. Left: First 1000 mini-batch updates, most peaks co-occur. Middle: Training throughout 10000 mini-batch updates. Right: Relationship between FCD evaluated with $m = 128$ and $m = 10^4$ fake samples, and the best linear fit $r = 0.996$.

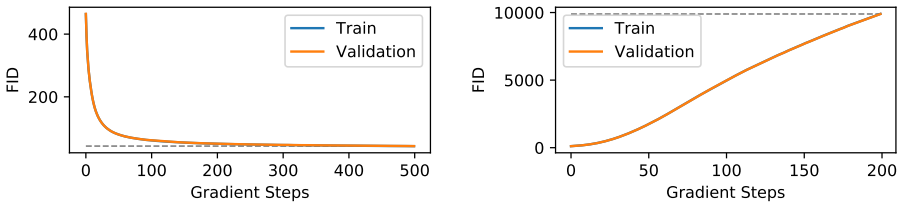
5 Adversarial Examples

Previous work showed that it is possible to make images that fool neural network classifiers [13]. One well know example makes an imperceptible modification to a correctly classified input image of a panda by adding a small vector of "noise," which fools the classifier to predict the new input as a gibbon with high confidence [13].

What would adversarial examples be for FID? Here, we present two types. First, images that perceptibly look like random noise but attain good FID, denoted "adversarial noise for FID." For an example, see the second row of Figure 1a. Second, images that perceptibly look real but attain a bad FID, denoted "adversarial examples for FID." For an example, see the second row of Figure 1b. Together, these two types of adversarial attacks show that it is possible to fool the FID.

A simple way to make an adversarial input, being noise or a natural looking image, is to optimize $FID(z)$ by gradient updates from some starting point z. We take z to be samples from either random noise vectors $z \sim N(0, 1)$ or random samples from the CelebA dataset [16]. To ensure that generated samples remain valid, i.e., in the range $[0; 1]$, under gradient updates, we apply the sigmoid function to the adversarial noise and clip values of adversarial examples. To compute FID during training, we have pre-computed that statistics for a training set of 190 thousand samples and a validation set of ten thousand samples. We optimize the same batch of 64 samples to either maximize or minimize FID. Note that the reported FID_{64} is biased in the sense that it is an upper bound of FID_{10^4}. This bias means that the adversarial noise may have a better (lower) FID_{10^4} since it is no more than FID_{64}. On the other hand, it means that the adversarial examples may be worse (lower).

Adversarial Noise for FID. Figure 6a depicts the training and validation loss during 500 gradient steps on a batch of noise. The validation loss reaches an FID of 42.6 after 500 steps, as indicated by the gray dashed line. In Figure 1a, one can compare random samples from the initial random batch of



(a) Adversarial Noise (b) Adversarial Examples

Figure 6: FID loss curves for training adversarial examples. Validation and training loss are so close the lines overlap. (a) is loss corresponding to the fakes of Figure 1b and right plot is loss for Figure 1a.

noise, which has a validation FID of 463.9, and the same samples but after the 500 gradient steps. The produced adversarial noise still visually looks like random noise but attains a low FID.

Adversarial examples for FID. Figure 6b shows the training and validation loss during 200 gradient steps on a random sample of 64 real images from the CelebA data set. After 200 steps, the validation FID is 9901.1, as indicated by the dashed grey line. Again, for a comparison between original samples and optimized samples, see Figure 1b.

6 Related Work

The literature contains several methods for evaluating generative models. An overview can be found in [4] which reviews a total of 29 different methods. This article concerns the FID, which is arguably the most popular method, often used to benchmark state of the art generative models like, e.g., [5].

6.1 Computational Resources

To the best of our knowledge, there exists no published article which reduces the computational resources associated with the FID. However, some open-source implementations of FID contain modifications which remove parts of the resource heavy computations. Such "modified FIDs" are faster but resolves to approximations. Below, we compare two modifications with our algorithm.

[9] reduces time consumption from $O(d^3+d^2m)$ to $O(dm)$ by computing only $||\mu_1-\mu_2||_2^2$, omitting the trace term $\text{tr}(\Sigma_1+\Sigma_2-2\sqrt{\Sigma_1\Sigma_2})$. [7] incorporates the covariance matrices while retaining the $O(dm)$ time complexity, by only using the diagonal entries of the covariance matrices. We emphasize that leaving out parts of the FID formula can introduce arbitrarily high levels of error. In comparison, our algorithm computes the exact FID in $O(d^2m)$ time. Even though our algorithm has a higher asymptotic time complexity, for $m \leq 128$ it consumes no more than double the time of [7, 9]. This is true because both algorithms compute Inception encodings which take more than half of the computation time.

Finally, [8] computes $\text{tr}(\sqrt{\Sigma_1\Sigma_2})$ by computing the eigenvalues of $\sqrt{\Sigma_1}\Sigma_2\sqrt{\Sigma_1}$. This has the advantage that Σ_1 is symmetric so one can compute $\sqrt{\Sigma_1}$ through its eigendecomposition. Such approach is not possible for $\Sigma_1\Sigma_2$ which, in general, is not symmetric. The derivations of [8] have some similarities to our derivations, but they do not exploit small m and thus take $O(d^3+d^2m)$ time.

6.2 Adversarial Examples

The Inception Distance (ID) [22] was widely used before the introduction of the FID. Previous work has successfully constructed adversarial examples against ID [2]. To the best of our knowledge, such adversarial examples have not been constructed for the FID. FastFID allows constructing such adversarial examples efficiently by simple gradient descent as presented in Section 5.

7 Conclusion

We introduced FastFID which computes FID fast when the number of "real" samples is large and the number of "fake" samples is small. Depending on the number of "fake" samples, FastFID is between 25 to 500 times faster than a previous implementations. FastFID can be extended to the Frechet ChemNet Distance, FastFCD, which is between 30 to 150 times faster than a previous implementation. The attained speed-up is useful when evaluating FID/FCD frequently during training. Finally, FastFID allow the construction of adversarial examples for FID by simple gradient descent.

Broader Impact

Potential Positive Impact. FastFID allow researchers to quantitavely test the training of generative models faster than previously possible. This is particularly useful for researchers with a constraint on their computational budget, e.g., a PhD student with a single GPU will benefit more than a industry lab with several GPUs. We hope this decreases the computational budget needed to perform experimental research for generative models.

Potential Negative Impact. FastFID allows one to efficiently cheat FID. This could make FID less reliable, since bad actors can use FastFID to train their generative model to make adversarial examples for FID. For example, a bad actor might first train a competitive generative model, then fine-tune the model to attain state of the art performance by directly minimizing FID.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[2] Shane Barratt and Rishi Sharma. A Note On the Inception Score. *arXiv preprint arXiv:1801.01973*, 2018.

[3] Åke Björck and Sven Hammarling. A Schur Method for the Square Root of a Matrix. *Linear algebra and its applications*, 1983.

[4] Ali Borji. Pros and Cons of GAN Evaluation Measures. *Computer Vision and Image Understanding*, 179:41–65, 2019.

[5] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large Scale GAN Training for High Fidelity Natural Image Synthesis. In *International Conference on Learning Representations*, 2019.

[6] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing Neural Networks with the Hashing Trick. In *International conference on machine learning*, pages 2285–2294, 2015.

[7] TensorFlow Contribute. Diagonal FID. `https://github.com/tensorflow/tensorflow/blob/00fad90125b18b80fe054de1055770cfb8fe4ba3/tensorflow/contrib/gan/python/eval/python/classifier_metrics_impl.py#L582`.

[8] TensorFlow Contribute. Eigen FID. `https://github.com/tensorflow/tensorflow/blob/00fad90125b18b80fe054de1055770cfb8fe4ba3/tensorflow/contrib/gan/python/eval/python/classifier_metrics_impl.py#L411`.

[9] TensorFlow Contribute. Mean FID. `https://github.com/tensorflow/tensorflow/blob/00fad90125b18b80fe054de1055770cfb8fe4ba3/tensorflow/contrib/gan/python/eval/python/classifier_metrics_impl.py#L525`.

[10] Edvin Deadman, Nicholas J Higham, and Rui Ralha. Blocked Schur Algorithms for Computing the Matrix Square Root. In *International Workshop on Applied Parallel Computing*, pages 171–182. Springer, 2012.

[11] DC Dowson and BV Landau. The Fréchet Distance between Multivariate Normal Distributions. *Journal of multivariate analysis*, 1982.

[12] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets. In *NIPS*, 2014.

[13] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. In *ICLR*, 2015.

[14] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium. In *Advances in neural information processing systems*, pages 6626–6637, 2017.

[15] Alex Krizhevsky. Learning Multiple Layers of Features From Tiny Images. Technical report, 2009.

[16] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep Learning Face Attributes in the Wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[17] Yuji Nakatsukasa. The Low-Rank Eigenvalue Problem, 2019.

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*, 2019.

[19] Daniil Polykovskiy, Alexander Zhebrak, Benjamin Sanchez-Lengeling, Sergey Golovanov, Oktai Tatanov, Stanislav Belyaev, Rauf Kurbanov, Aleksey Artamonov, Vladimir Aladinskiy, Mark Veselov, Artur Kadurin, Sergey Nikolenko, Alan Aspuru-Guzik, and Alex Zhavoronkov. Molecular Sets (MOSES): A Benchmarking Platform for Molecular Generation Models. *arXiv preprint arXiv:1811.12823*, 2018.

[20] Kristina Preuer, Philipp Renz, Thomas Unterthiner, Sepp Hochreiter, and Günter Klambauer. Fréchet ChemNet Distance: a Metric for Generative Models for Molecules in Drug Discovery. *Journal of chemical information and modeling*, 58(9):1736–1741, 2018.

[21] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In *4th International Conference on Learning Representations, ICLR*, 2016.

[22] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in neural information processing systems*, pages 2234–2242, 2016.

[23] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015.

[24] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 2020.