# A comparative evaluation of gradient-based optimization algorithms for training Terrain GAN

Kai Qin
*Dept. of Electrical and Computer Engineering*
*University of Texas at Austin*
Austin, TX
kai.qin@utexas.edu

Yi Han
*Dept. of Electrical and Computer Engineering*
*University of Texas at Austin*
Austin, TX
yh5598@utexas.edu

## I. Introduction

Generative Adversarial Networks (GANs) [1] have been wildly used in applications such as anime character generation, human-face generation, and video generation. In this study, we focused on evaluating the performances of three different gradient based optimizers on our TerrainGAN, explored the practical differences from using each optimizers, and discuss the results of fine-tuning hyper-parameters and visualize their effect on our GAN.

## II. Gradient-based optimization algorithms review

Before we dive into the evaluation of three gradient-based optimizers we compared in this study, SGD with momentum, RMSProp, and ADAM, we want to provide a review of these three algorithms. We will start from the classic gradient descent algorithm. In one sentence, the general idea of the classic gradient descent algorithm is that at every iteration, the desired parameters are moving in the direction of the negtive gradient of the objective function based on the entire training dataset to decrease the loss function. The vanilla gradient descent may be accelerated considerably by using stochastic gradient descent (SGD) which follows the gradient of randomly selected minibatches. Even though SGD ingtroduced this very important idea of training with minibatches, learning with it can sometimes be very slow. For example, let's say that you're trying to optimize a cost function which has a contour in image 1. The red dot denotes the position of the minimum. And we can see this up and own oscillations in black lines slows down the gradient descent. Therefore, it's rear to see large scale neural network training with classic SGD. One improvement made to SGD is by adding momentum.The basic idea of SGD with momentum is to compute an exponentially decaying moving average of the past gradients and continues to move in that direction [3]. The velocity update step in algortihm 2 shows how to compute the exponentially weighted averages of the last N iterations, where N is determined by the hyperparameter alpha. For exambple, when alpha equals 0.99, which is commonly used in practice, we are approximately averaging over the last 100 iterations. This method can only approximate the average value over the last N days, but it takes very little memory. Image 1 illustrates the effect of momentum.

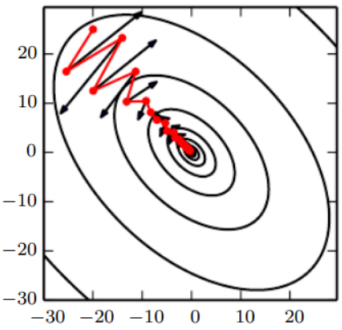


Fig. 1. Visualization of the effect of momentum [3]

**Require:** Learning rate $\epsilon$, momentum parameter $\alpha$.
**Require:** Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
**while** stopping criterion not met **do**
Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
**end while**

Fig. 2. SGD with momentum [3]

If you average out these gradients of the black arrows, you can find that the oscillations in the forward diagonal direction will tend to average out to something closer to zero. In the mean time, it will keep the direvative on the backward diagonal direction. So this allows the optimizer to take a more straight foward path or to damp out the oscillations in the path to the minimum.

Root mean squared prop (RMSProp) is another algorithm that can speed up the classic gradient descent. This algorithm (see figure in 3) is very similar to SGD with momentum, but instead of accumulating the gradient, we are taking the average of the sqaured gradient and divide the gradient by the sqaure root of the average squared gradient [3]. By dividing the average magnitude of the derivative in each dimenstion, the net effect is that the forward diagonal derivative in image 1 is devided by a relatively larger number, and it therefore helps damp out the oscillations. Comparing to SGD with

**Require:** Global learning rate $\epsilon$, decay rate $\rho$.
**Require:** Initial parameter $\boldsymbol{\theta}$
**Require:** Small constant $\delta$, usually $10^{-6}$, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
**while** stopping criterion not met **do**
Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
**end while**

Fig. 3. RMSProp [3]

**Require:** Step size $\epsilon$ (Suggested default: 0.001)
**Require:** Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
**Require:** Small constant $\delta$ used for numerical stabilization. (Suggested default: $10^{-8}$)
**Require:** Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
Initialize time step $t = 0$
**while** stopping criterion not met **do**
Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
**end while**

Fig. 4. ADAM [3]

momentum, we can use a larger learning rate, and get faster learning without diverging in the forward diagonal direction, where in SGD with momentum, the learning rate is applying to all each dimension with same step size.

By combining both ideas of momentum and RMSProp, we get Adam optimization algorithm which has been shown to work well across a wider range of deep learning architectures. In Adam, in each iteration, in the first highlighted equation in figure 4 we first update the first moment estimate (the mean of the derivatives) , which is exactly what we had when we're implementing SGD with momentum. And similarly, we do the rms prop to update the second moment estimate in the second equation. Then we do bias corrections. Finally, we compute the update by dividing the bias corrected first moment estimate with the sqaure root of the second moment estimate and reversing the sign. Common choices for $\rho_1$, $\rho_2$, $\sigma$ are 0.9, 0.99 and $10^{-8}$ respectively. In this study, we followed the common pratice for Adam hyperparameter tunning which keeps $\rho_1$, $\rho_2$, and $\sigma$ as default and try a range of values of the learning rate to see which works the best.

## III. Terrain GAN review

Traditionally, video game terrains have been either manually generated or procedually gendreatd by algorithms designed to mimic real-life terrains such as mountains, lakes, and coasts.

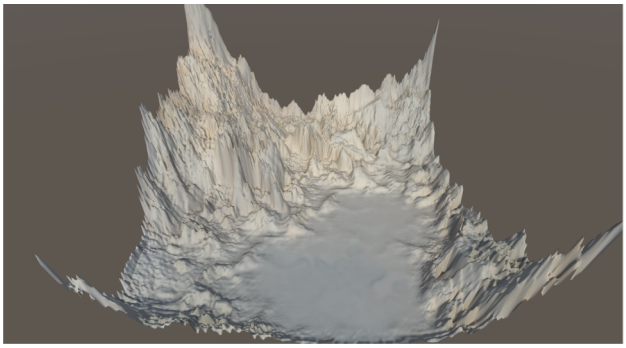

Fig. 5. A redering of a generated heightmap in [2]

These methods are capable of generating high quality terrains but also come with drawbacks such as high expense of human labour and the lack of flexibility and complexity of the nature. With the ability to recover the training data distribution [1], GAN becomes the perfect algorithm for this task. A first step towards conquering this problem using GAN has been proposed in [2], where DCGAN and LSGAN have been applied to generate high quality heightmaps (see image 5) from high-resolution earth heightmap data provided by NASA. In our project, we focus on evaluating the effects of different optimizers on LSGAN since it is suggested in the paper [2] that LSGAN provide better training stability than DCGAN. One novelty of GAN is learning the training data distribution via an adversaria process. The training process consists of two steps as shown in the psudocode in figure 7, where we first train the discriminator$G$ for $k$ steps, and then train the generator $D$ for one step. The number of steps to apply to the discriminator, $k$, is a hyperparameter. As in the GAN paper, we used $k = 1$, the least expensive option, in all of our experiments in the next section.

DCGAN and LSGAN are two variants of GAN, where DCGAN improves the orginal GAN with fine-tuned architecture details and showed us that GAN is capable of generating perceptually good samples (see figure 6 and interpolations [4]. And LSGAN adopted the least squares loss instead of the cross entropy as the objective function since the original loss function may lead to the vanishing gradients problem.

## IV. Experiments and results

In this project, the training data are 128px height maps from the original NASA image As we can see in the graph on the right, the loss function of both G and D oscilate and don't converge. That's one of the big challenge of GAN training is that you don't get this clean monotonic improvment that your are used to with training supervised models.In GANs, the objective function for the generator and the discriminator usually measures how well they are doing relative to the opponent. For example, we measure how well the generator is fooling the discriminator.

Sometime it oscilates and don't converge. That's one of the big challenge of GAN training is that you don't get this clean

Fig. 6. DCGAN samples on faces [3]

**for** number of training iterations **do**
  **for** $k$ steps **do**
  - Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of $m$ examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

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

  **end for**
  - Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

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

**end for**
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Fig. 7. Minibatch stochastic gradient descent training of GAN [1]

monotonic improvment that your are used to with training supervised models.

"It is still an open probelm to have good metrics. If you had a really really good metric, you can propabally use it as a optimiztion critieria and optimize against it" "Assuming you don't do optimization directly on the Frechet Inception Distance, it can be a nice independent measure of the amount of variantion and cristness of the image generated ; when you do optimize against it, you will find results not exactly what you want" Adam 0.0002 lr and 16 batch size, we think based on the 3d model generated by this software looks realistic enough to meet our experience. We manually sweeped the lr rate and batch size using ADAM and collected the loss fuction over trainnign iterations. Next step will colecte results using other optimizers and figure out a metric to quantify the results we see. The discriminitor can always

## V. Discussion about metrics

As we have seen in the loss graphs, the oscilation of the loss makes it harder to measrue the training progress. Currently, it is still an open probelm to find good metrics for GAN training. On the other hand, If you had a really really good metric, you can propabally use it as a optimiztion critieria and optimize against it. Inception score [5] and Frechet Inception Distance score (FID) [6] are two state of the art metrics

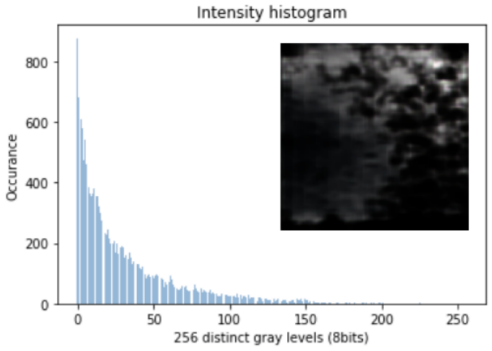


Fig. 8. Image intensity histogram of a generated height map.

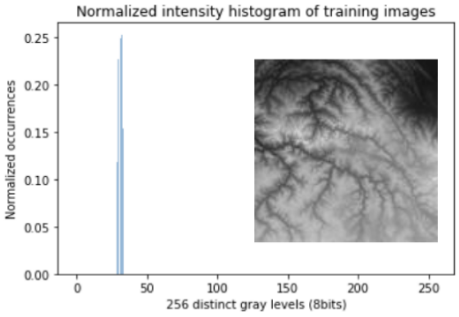


Fig. 9. Normalized image intensity histogram of the training dataset and an example training image.

people use to measure the GAN performence. However, both metrics don't apply to the training data we have. Inception score requires categorical labeled data. FID requires a pre-trained classification neural network. Inspired by the goal of gan which is to learn the probability distribution from the training data, and implicitly represent it using neurual network, we propose a new metric to measure the Terrain GAN performance, the KL-divergence between normalized intensity histogram of training images and generated images.

Before we explain how we calculate this metric, we need first introduce the idea of image intensity histogram, which is a widly used technique in image processing. It plots how many times each intensity value in an image occurs. Here we have a example of intensity histogram of a generated hight map in figure 8. However, we want the distribuion over a large set of images, the original intensity historgram doesn't work here, so we calculate the average occurrences of each intensity value and divide it by the total number of pixels over all images, and we call it the normalized histogram. The normalizatoin also remaps the occurences to the same range $[0, 1]$ as the probability distribution. Image 9 shows the normalized itensity histogram of the training images, and image 10 shows the normalized itensity histogram of 50 generated image using the neural net trianed by Adam with learning rate of $0.0002$ and batch size of 16 for 150 epchos.

Given the normalized image intensity histograms of the training data and generated samples, we can calculate the KL divergence using equation (1), where $P(x)$ is the training data

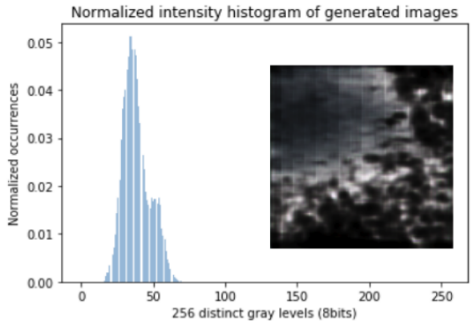


Fig. 10. Normalized image intensity histogram of generated images from LSGAN trained by ADAM with $lr = 0.0002$ and $batchsize = 16$ and a sample image. KL divergence of normalized intensity histogram equals 1.62

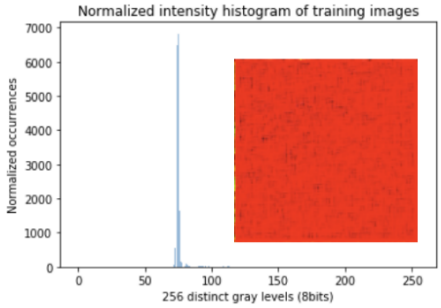


Fig. 11. Normalized image intensity histogram of generated images from LSGAN trained by SGD with $lr = 0.0001$ and $batchsize = 16$ and a sample image. KL divergence of normalized intensity histogram equals $+\infty$

distribution and $Q(x)$ is the generated sample disctribution. Using the training dataset normalized histogram in image 9 as a reference distribution, image 10 shows a normalized histogram with KL-divegence of 1.62 and image 11 shows a normalized histogram with KL-divergence of $+\infty$. We get a positive infinity since the training data and the generated data in figure 11 have no overlap in normalized histogram.

$$D_{KL}(P||Q) = \sum_{x \in \chi} P(x) log(P(x)/Q(x)) \qquad (1)$$

This metric has been utilized in measuring the quality of different hyperparameters when use Adam to train LSGAN. Figure 12 shows a bar plot of the KL divergence of different hyperprameter settings of Adam after 150 ephchs. Visually, we can see that the higher the metric, the harder you can provide some geographic meaning to the generated images.

## VI. Conclusion

## VII. Future Works

## References


[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[2] C. Beckham and C. Pal, "A step towards procedural terrain generation with gans," 2017.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.


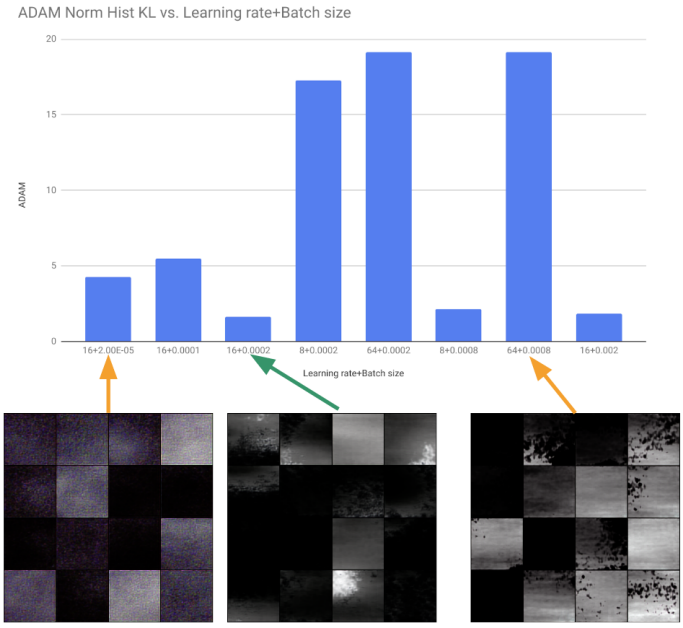


Fig. 12. Normalized histogram KL divergence of generated samples vs. different Adam learning rate and batch size


[4] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2015.

[5] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," 2016.

[6] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," 2017.