Terrain GAN optimizer study

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. Terrain GAN

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. 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 1). 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.

Following [2], we found an open sourced NASA grayscale topography map of Earth. The value of the each tile is grayscale, ranging from 0 to 255. Higher value in the grayscale represents higher altitude such as mountain tops, while lower values represent valley or sea level. The 21600x10800 pixels high resolution map is subsequently divided into 128x128 pixel tiles, which provides about 14200 training data for our GAN. However, since about 70% of Earth is water and therefore uninteresting in the purpose of generating terrain with variations, tiles that are purely sea level are removed, leaving about 4300 training tiles fit for training. 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 3, 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.

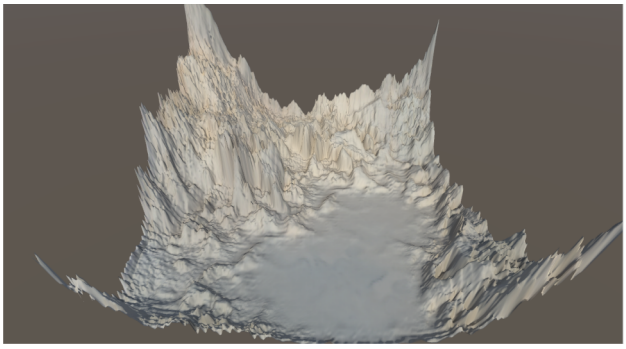

Fig. 1. A redering of a generated heightmap in [2]

Fig. 2. DCGAN samples on faces [3]

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 2 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.

III. 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

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. 3. Minibatch stochastic gradient descent training of GAN [1]

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 4. 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 5 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 4 illustrates the effect of momentum. 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 6) 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,

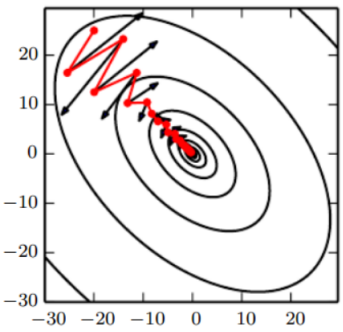

Fig. 4. Visualization of the effect of momentum [3]

Require: Learning rate ϵ, momentum parameter α.
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. 5. SGD with momentum [3]

the net effect is that the forward diagonal derivative in image 4 is devided by a relatively larger number, and it therefore helps damp out the oscillations. Comparing to SGD with 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 7 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

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, 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. 6. RMSProp [3]

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
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. 7. ADAM [3]

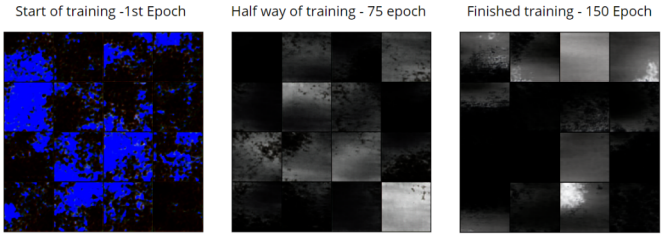

Fig. 8. Sample generated output from different training stages

and reversing the sign. Common choices for ρ_1, ρ_2, σ are 0.9, 0.99 and 10^{-8} respectively. In this study, we followed the common pratice for Adam hyperparameter tunning which keeps ρ_1, ρ_2, and σ as default and try a range of values of the learning rate to see which works the best.

IV. Experiments and Results

All experiments of TerrainGAN takes 150 epochs, which means 150 iterations through the entire 4300 training images, to complete. Generator outputs throughout the training process are periodically saved to visualize convergence and success. Generator/discriminator parameters, loss function plot, and training time are also saved for further analysis. Training was completed on a 2080Ti GPU and 32GB of RAM for hardware reference. We purposefully imported the grayscale images as colored RGB images into the training dataset to provide an extra way of visualize optimizer performance. Good optimizers will eliminate color at the very beginning of training. Similarly, we did not change input latent variable size from 100 to a number that is a power of 2 to visualize the checkerboard effect [?], which signals inferior convergence.

Fine tuning adam optimizer

The most visually diverse and successful result was ran using the Adam optimizer with learning rate 0.0002 and batch size 16. As we can see in the loss function graph on the right, the loss function of both generator G and

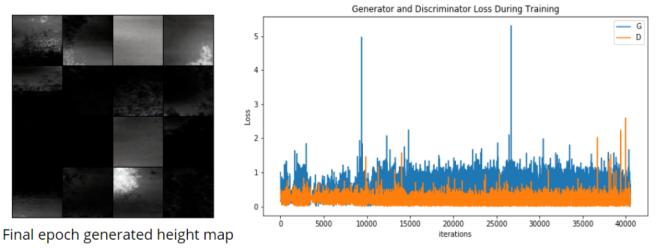

Fig. 9. Sample output from Adam optimizer (left), loss function graph of generator and discriminator (right)

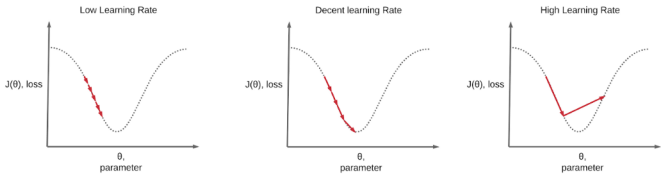

Fig. 10. 2-dimension visualization of learning rate effects

discriminator D oscillate and don't converge. One of the bigger challenges in GAN training is that you don't get a clean monotonic improvement curve produced 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. We confirm how well the generator is fooling the discriminator by observing a consistent oscillating behavior. The final epoch produced a batch of sample outputs that is very representative of the training dataset, which includes visual geographical features such as mountains, coast, and deltas.

Adam hyperparameter fine-tuning gave a great visualized reinforcement of lessons learning in class about optimizers. The two hyperparameters with the most effect on training our TerrainGAN are learning rates and mini-batch sizes. The importance of tuning learning rate and mini-batch size to expedite convergence and decrease training time can be visualized in the next two sections. When taught the importance of learning rate(lr) in class, a 2 dimension graph comparison of low/correct/high learning rate is often shown. We define best lr = 0.0002 from previous runs, and correspondingly ran three training runs of TerrainGAN with 0.1*lr, lr, and 10*lr. We found close correlation between the 2 dimension graphs and our training process under these learning rates. Throughout the training process, Adam with 0.1*lr is struggling to even converge on producing grayscale tiles, visualizing the effect of using a small sub-optimal learning rate.

Adam with 10*lr will occasionally produce red shaded tile outputs, from otherwise a mostly converged generator. This phenomenon precisely visualizes the effect of big learning rate jumping out of the local minimum valley. Final epoch result also shows a less uniform distribution of pixels when compared to the best learning rate.

As one of the primary factors for optimizer convergence rate, it is best advised to try out a range of learning rates

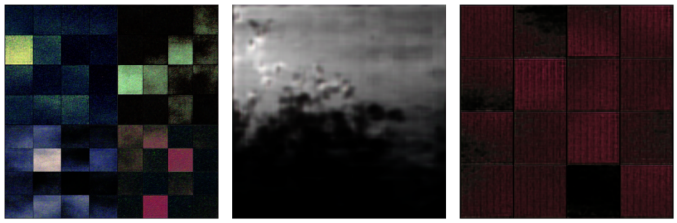

Fig. 11. Colorful output from learning rate too small (left), single feature rich heightmap from best learning rate (middle), red shaded output from learning rate too large (right)

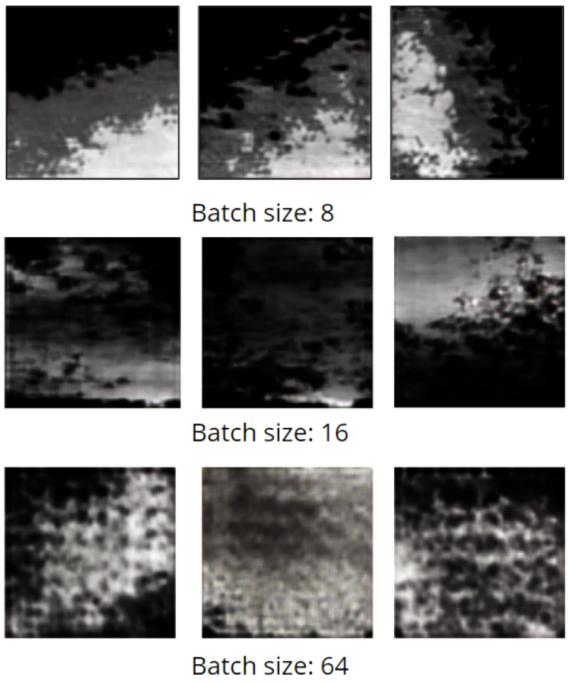

Fig. 12. Sample output from 3 different batch sizes

during the early stages of GAN training process and examine the results, before committing heavy computational resources.

Training time, overfitting, and GAN convergence are all closely tied to batch size tuning, as our training trials suggests. We used batch size 8, 16(optimal), and 64 to display our fine tuning findings.

From some randomly selected output tiles of each batch size, we can clearly see overfitting with batch size 8, which only uses 3 different values(black, gray, white) of the 256 value grayscale to try to fool the discriminator. On the other end, using batch size 64 against a dataset with only 4300 training data proved too feature diverse for the generator to learn over the 150 epoch period. The output heightmaps leave checkerboard artifacts from unfinished deconvolution [?].

When using DCGAN, a less stable version of LSGAN that utilizes binary cross entropy loss, large batch size of 64 will consequently completely fail to converge during the initial epochs of training, reiterating the importance of batch size tuning for GAN.

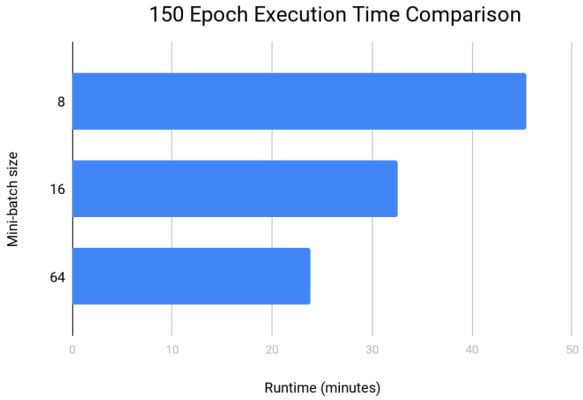

Fig. 13. Training time comparison between batch sizes

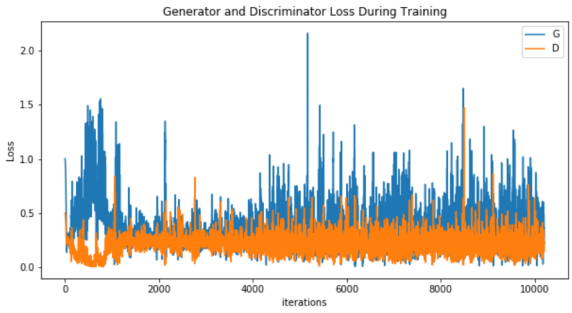

Fig. 14. Danger of complete divergence during initial epochs when using large batch sizes

Optimizers comparison

Given the same learning rate and batch size, we compared the performence of four optimizers-SGD, SGD with momentum, RMSProp, and Adam traing for the same number of epchos. Figure15 and **??** shows the results of 150 epchos of SGD, SGDM and ADAM and their loss graph over iterations. Visually, SGD fails generating any geographical features that we want such as mountains or lakes, valley, coasts. SGDM produced a largly reddish images which clearly shows that the output distribution is different from the training data. The generator loss of SGD and SGDM remains around 0.5, and the discrimintor stays around 0 which represents that the learning has not reached a point where generator is able to fool the discriminator. For adam graphs, this is another example of too big of learning rate which fail to converge.

V. Discussion about metrics

As we have seen in the loss function graphs, the oscillation makes it harder to measure the success of the training progress. Currently, it is still an open problem to find good metrics for GAN training. Simply put, If you have a validation metric, you can probably use it as the objective function and optimize against it. Inception score [5] and Frechet Inception Distance score (FID [6]) are two state of the art metrics people use to measure GAN performances. However, both metrics don't

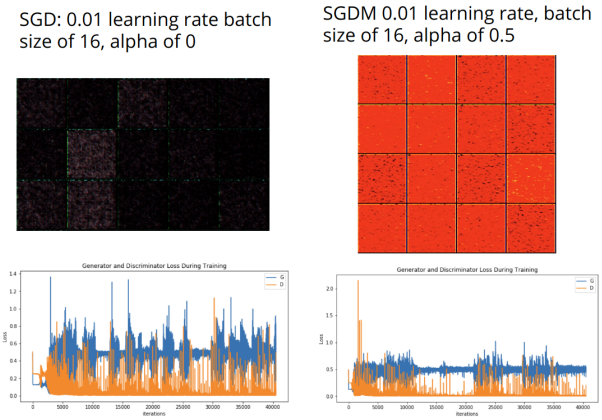

Fig. 15. SGD vs. SGD with momentum

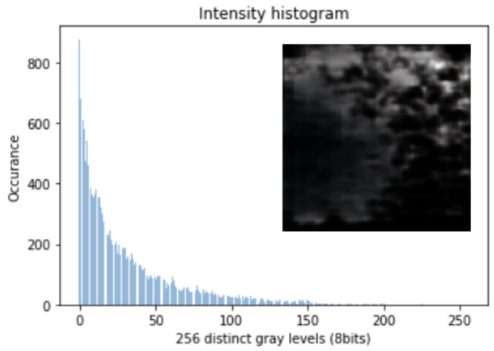

Fig. 16. Image intensity histogram of a generated height map.

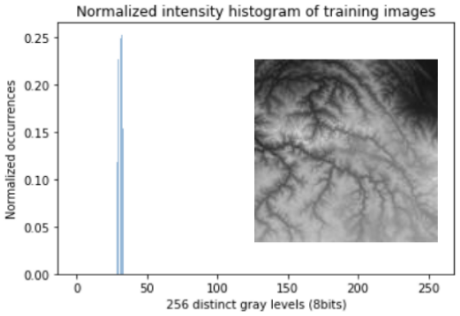

Fig. 17. Normalized image intensity histogram of the training dataset and an example training image.

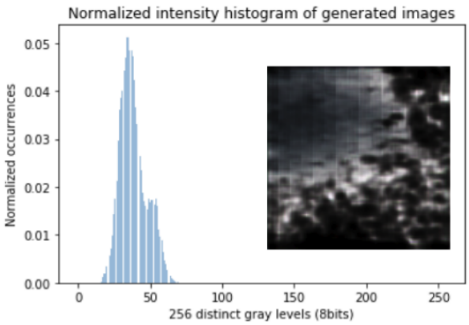

Fig. 18. Normalized image intensity histogram of generated images from LSGAN trained by ADAM with $lr = 0.0002$ and $batchsize = 16$ and a sample image. KL divergence of normalized intensity histogram equals 1.62

apply to our training data. Inception score requires categorically labeled data and FID requires a pretrained classification neural network. Inspired by the goal of GAN to learn the probability distribution from the training data and implicitly represent it using neural networks, we propose a new metric to measure the TerrainGAN performance: the KL-divergence between normalized intensity histogram of training images and generated images.

Before we present how we calculated this metric, we need first introduce the idea of image intensity histogram, which is a widely used technique in image processing. The definition states the histograms plots how many times each intensity value in an image occurs. Here we have a example of intensity histogram of a generated heightmap in figure 16. We then produce the distribution over a large set of images by calculating the average occurrences of each intensity value and divide it by the total number of pixels over all images, and defining it as the normalized histogram. The normalizatoin process also remaps the occurences to the same range $[0, 1]$ as the probability distribution. Figure 17 shows the normalized intensity histogram of the training images, and figure 18 shows the normalized intensity histogram of 50 generated image using the neural net trained by Adam with lr .0002 and batch-size 16 for 150 epochs.

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 distribution and $Q(x)$ is the generated sample disctribution. Using the training dataset normalized histogram in figure 17 as a reference distribution, figure 18 shows a normalized histogram with KL-divegence of 1.62 and image 19 shows a normalized histogram with KL-divergence of $+\infty$. We get a positive infinity since the training data and the generated data

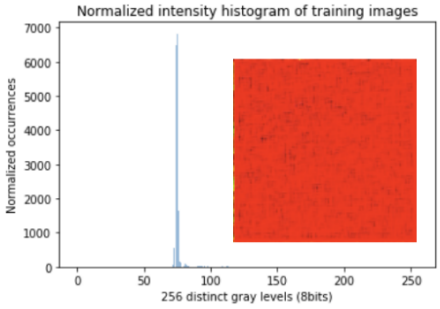

Fig. 19. Normalized image intensity histogram of generated images from LSGAN trained by SGD with $lr = 0.0001$ and $batchsize = 16$ and a sample image. KL divergence of normalized intensity histogram equals $+\infty$

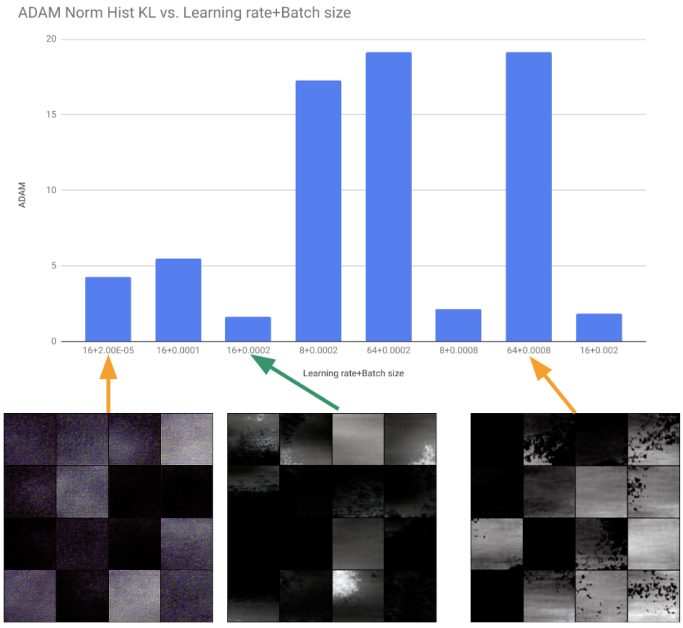

Fig. 20. Normalized histogram KL divergence of generated samples vs. different Adam learning rate and batch size

in figure 19 have no overlap in normalized histogram.

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

We used this metric in measuring the quality of different hyperparameters when use Adam to train TerrainGAN. Figure 20 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 geographic meaning to the generated images.

VI. Conclusion and Next Steps

In this optimizer study, we measured the performance of three different gradient based optimization algorithms used in our terrainGAN. We also highlighted the importance of choosing the right learning rate and batch size for proper GAN training. As a final thought on a month long training of GANs, look both visually and statistically at the generated data for metrics of success. More technical optimization method such as variable learning rates and noise injection will be next to tryout when training our future GANs. While working on TerrainGAN, we scoured information on the latest implementations of GANs such as styleGAN and progressiveGAN. We will be applying the lessons learned from this optimizer study into training art style transfer GANs and inpainting GANs.

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.
[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.