Optimization Methods for Conditional Generative Adversarial Networks: Training cGANs on Human-Drawn Sketched Images

Catherine Horng, ch756

Abstract

Conditional generative adversarial networks (cGANs) have had previous success in training generative models conditioned on class labels on datasets such as MNIST. The optimization techniques used to train these models can greatly affect the time to train and outcomes of these models. Here we explore the different optimization techniques stochastic gradient descent (SGD), SGD with momentum, and adaptive optimization methods (Adam, RMSProp, and AdaGrad) on cGANs. With the growing interest in GANs for generating images and art, the experiments utilize crowdsourced human-drawn sketched images to model the expanding diversity GANs have been used on in recent years. With these images, while training wall-clock time greatly varies with choice of optimizer and hyperparameter with no clear trend, training with RMSProp does better than the baseline SGD in statistical performance measures, with a 46.47% increase in inception score (IS) and a 27.75% decrease in Fréchet inception distance (FID) for the best trained model. Adaptive learning rate methods generally improve upon our baseline while other methods ultimately perform poorly.

1. Introduction

Generative adversarial networks (GANs) are a machine learning framework consisting of a generative and discriminative model trained in conjunction to compete via an adversarial process (Goodfellow et al., 2014). A common analogy for this framework is to view the generator as a forger who creates forgeries while the discriminator is an expert who aims to distinguish between forgeries and authentic examples (Creswell et al., 2018). Much of the recent research on GANs focuses on image synthesis which has resulted in realistic images that can fool even humans (Karras et al., 2018). While the original GAN framework proposed does not control what image is generated based on the original dataset, conditional GANs (cGANs) are a form of GANs in which the model can generate specific images conditioned on class labels; this type of GAN has been shown to be successful on the MNIST dataset, producing realistic digits (Mirza & Osindero, 2014).

Training cGANs and GANs in general can be difficult (Saxena & Cao, 2020). Because of the adversarial nature of the system, the training of the generator and discriminator must be synchronized to ensure that one player does not overpower the other, thus leading to an unstable system in which the entire system can not continue to learn (Kodali et al., 2017). For this reason, choosing an optimization strategy still remains a challenge for training GANs. Much like in other deep learning practices, different optimizers may result in varying results in both statistical and hardware performance. Common optimizers that can be used to train deep learning models including GANs include stochastic gradient descent (SGD), SGD with momentum, Adam, RMSProp, and AdaGrad; all of which have resulted in previous success (Ruder, 2017). However, there is no definitive best practice for choosing an optimization strategy for training cGANs as choosing optimizers and hyperparameters are extremely task dependent.

Recently there has been growing interest in introducing deep learning into art by means of generative modelling, particularly through the use of generative adversarial networks (Shahriar, 2021). This interest has been aided by the introduction of datasets such as Google's Quick, Draw![1], a crowdsourced dataset of sketches of various categories. Google's Quick, Draw! is visually similar to the popular MNIST dataset as both are vector based hand-drawn images, but Google's Quick, Draw! provides more diversity in each category, thus creating an arguably more difficult but more applicable task in reference to utilizing GANs to create art.

In this project, we construct a cGAN on Google's Quick, Draw! dataset and train it with various optimizers. We also minimally explore hyperparameters for each optimizer in order to determine what optimization strategy should be used in order to generate the most similar images to the Quick, Draw! Dataset.

[1] https://quickdraw.withgoogle.com/data

2. Background

2.1. GANs and cGANs

Research into generative models have expanded in recent years, as has its interest in being applied to different applications (Saxena & Cao, 2020). Generative models aim to compute a new data distribution by approximating the true data distribution and GANs have emerged to be a powerful method to creating these generative models by utilizing a discriminator to distinguish between real and fake examples so that the generator is forced to learn to generate samples more similar to the data it is being trained on (Goodfellow et al., 2014). At its core, simple GAN models consist of two neural networks that compete with each other to optimize their individual loss functions in the zero-sum game to find the global Nash equilibrium (Salimans et al., 2016).

Conditional GANs (cGANs) are GANs in which both the generative and discriminative model are conditioned on some extra information fed in as input data (**?**); in our case, this extra information is the class label so that we can produce samples of a particular class.

2.2. Training GANs

Training GANs have proven to be an ongoing challenge; GANs often find it difficult to generate samples with increasing diversity; this results in the mode collapse problem in which the generative model fails to produce any examples of a certain class (Mirza & Osindero, 2014). Additionally, the generative and discriminative networks must train at the same rate to avoid one network overpowering the other and thus resulting in a case in which neither network continues to train (Salimans et al., 2016).

2.3. Quick, Draw! Dataset

Google's Quick, Draw! Dataset is a collection of 50 million drawings over 345 categories. The dataset was constructed by players of the game Quick, Draw! by Google in which players are given a category to draw in 20 seconds. Each sample in the dataset has been preprocessed and rendered into a 28×28 grayscale numpy bitmap file, similar to the MNIST dataset. However, while in the case of the MNIST dataset, differences within a class can be small (a drawn number "4" can visually look very similar to another drawn number "4") this is not the case for the Quick, Draw! dataset. A drawn image of an airplane can vary greatly across different players as can a drawn image of an apple, for example. This results in a far more diverse dataset than the MNIST dataset which more closely mirrors the diversity we would expect to see in art.

2.4. Optimization Methods

2.4.1. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) uses one sample randomly to update the gradient per iteration instead of calculating the exact value of the gradient (Ruder, 2017); because the stochastic gradient is an unbiased estimate of the real gradient, this algorithm is known to converge (Ruder, 2017). Additionally, it has shown to reduce computational complexity and can converge quicker than traditional gradient descent. This method is the baseline for all other methods we will compare to.

As an extension of SGD, momentum is a method which accelerates the gradient vectors in the right direction (Qian, 1999). This is done by adding a fraction to the updated weights so that when the gradients remain in the same direction, updates are increased and when the gradients change direction, updates are reduced, thus granting us faster convergence and reduced oscillation. This way, momentum stores an exponential decaying average of past gradients and utilizes it to determine by how much each step should move in the direction of the gradient.

2.4.2. Adagrad

Adagrad is a method to adapt the learning rate by performing smaller updates based on historical gradients (Duchi et al., 2011). The gradients are accumulated and used to adjust the learning rate in each iteration. AdaGrad automatically tunes the learning rate but the global learning rate still requires tuning, and as the model continues to be trained, the learning rate will tend to zero, and thus the weights will not continue to update.

2.4.3. RMSProp

RMSProp was proposed as a solution to the tendency of the learning rate to go to zero in AdaGrad (Ruder, 2017). Rather than utilizing all historical gradients, RMSProp utilizes an exponential decaying average of past squared gradients. Additionally, there is no need for a default learning rate as it is no longer needed in the update rule. This method is almost identical to Adadelta.

2.4.4. Adam

Adaptive moment estimation (Adam) is again a method to compute adaptive learning rates (Kingma & Ba, 2017). It stores an exponential decaying average of past squared gradients, similar to RMSProp. It also stores an exponentially decaying average of past gradients as for momentum. Because of this, Adam can be seen as a combination of RMSProp and momentum.

3. Methods

3.1. Optimization Methods

We test five optimization methods: stochastic gradient descent (SGD), SGD with momentum, AdaGrad, RMSProp, and Adam. For each optimization strategy we do a simple hyperparameter exploration via grid search for the learning rate, α, having α vary between 0.1, 0.01, 0.005, 0.001, 0.0005, 0.0001, and 0.00001. Additionally, for SGD with momentum, we let the momentum hyperparameter β be 0.9 or 0.99. This results in a total of 42 models trained.

RMSProp has the additional hyperparameter ρ, which is the discounting factor for the history gradient. This is set to the default value of 0.9. Adam has the additional hyperparameters β_1 and β_2. These are set to the default values of 0.9 and 0.999.

We compare these models to the baseline of a cGAN trained with SGD as the optimization method with the traditional standard learning rate of 0.1.

Each model is trained to convergence for 20 epochs on a subset of the Quick, Draw! dataset, comprised on 10 classes, each with 4000 images in each class. The classes are 'airplane', 'apple', 'bus', 'fish', 'key', 'nose', 'snake', 'purse','star', and 'tree'. An example of each class is shown in Figure 1.

Figure 1. Examples of each of the classes from Google's Quick, Draw! Dataset.

3.2. Model Architecture

Because of the success convolutional neural networks have demonstrated on image data (Creswell et al., 2018), we use these networks to build the GAN generator and discriminator. Building off the conditional GAN from the Keras code example in generative deep learning [2], the generator consists of a dense layer with LeakyReLU activation, two deconvolution layers, each with LeakyReLU activation and a final convolution layer with a sigmoid activation and the discriminator consists of two convolution layers, each with LeakyReLU activation and a final global max pooling and dense layer.

[2]https://keras.io/examples/generative/conditional$_gan$/

3.3. Evaluation

Evaluating GANs is another challenge in and of itself (Borji, 2018). In evaluating the generation of synthetic images from GANs, human judgement has been the most reliable metric (Borji, 2018). While we will evaluate using visual inspection, we will also use quantitative statistical scores as well as wall-clock time to evaluate the hardware performance. While training, we can also measure the loss of the generative and discriminative models, however the losses of these models often are not indicative of the true performance, thus we omit reporting them in evaluating performance.

3.3.1. Hardware Performance

Wall-clock training time is used to evaluate the hardware performance of training the cGANs with different optimizers. Each model was trained for 20 epochs and the wall-clock time of training was measured. The models were trained on Google Colab with the Nvidia Tesla K80 GPU.

3.3.2. Statistical Performance

Inception Score: The Inception Score (IS) is likely the most commonly used score for GAN evaluation. It utilizes a pre-trained neural network to capture how classifiable and diverse generated images are with respect to the class labels (Salimans et al., 2016). Traditionally, in the evaluation of real images, the calculation of IS is implemented using Inception Net (thus the name of the score) trained on ImageNet, but for the purposes of evaluating hand-drawn images from the Quick, Draw! Dataset, the calculation of IS is implemented using a classification CNN trained on the Quick, Draw! Dataset. Generated images more similar to the real distribution of images will result in a higher IS, so we would like to find which optimizers produce a high IS. The minimum for this score is 1.

Fréchet Inception Distance: Fréchet Inception Distance (FID) embeds generated examples into a features space given by a layer of a CNN and views the embedding as a continuous multivariate Gaussian to estimate the mean and covariance of the generated and real examples (Heusel et al., 2018). The distance of these two Gaussians is the Fréchet distance. Similarly to IS, FID is implemented using Inception Net, but again for the purposes of our evaluation, we use the same classification CNN trained on the Quick, Draw! Dataset. Generated images more similar to the real distribution of images will result in a lower FID score, with the minimum score being 0, so we would like to find which optimizers produce a small FID score.

Classification Model: While the Quick, Draw! classifier is not the focus of this paper, because our evaluation metrics are contingent on it, it would seem to be beneficial to discuss the model accuracy and loss. The classifier consists of two layers of a convolution layer with ReLU activation and max pooling layer and two layers of a fully connected layer and a dense layer. The classifier trained to 98.94% accuracy on the training data and 88.58% accuracy on the test data. Because this classifier does not classify all input data correctly, the IS and FID scores may be deflated.

4. Results

4.1. Wall-Clock Time

The wall-clock times for each optimization method is shown in Table 5. A plot of the wall-clock time as a function of learning rate for each of the optimization method is shown in Figure 2. The baseline SGD with a learning rate of 0.1 trained in 1143.6545 seconds.

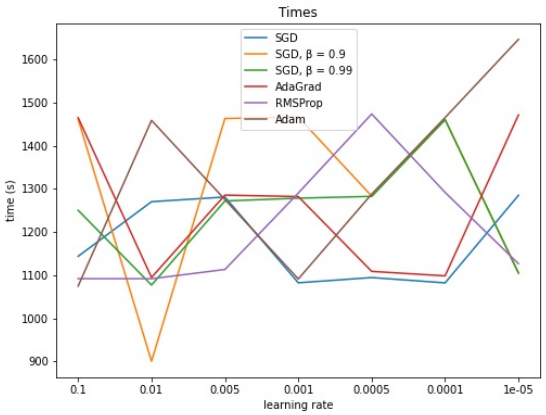


Figure 2. Graph of wall-clock training time for each optimization method varying learning rates.

It seems that no optimization method results in overall lower or higher wall-clock training time. Each model training with different hyperparameter values result in a varying wall clock times, with no clear trend.

4.2. Inception Score and Fréchet Inception Distance

The Inception Scores and Fréchet Inception Distances for each optimization method are shown in Table 6. The baseline SGD with a learning rate of 0.1 achieves an IS of 2.5631 and a FID of 0.4136.

Plots of the IS and FID as a function of learning rate for each of the optimization methods is shown in Figures 3 and 4. First we note that a lower IS generally corresponds to a higher FID and a higher IS corresponds to a lower FID, indicating that IS and FID are indeed measuring some latent quality in the images generated.

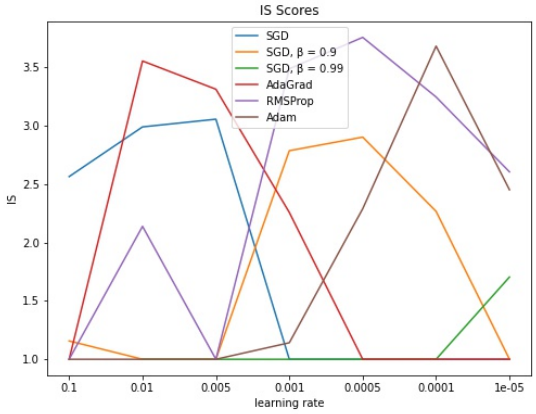


Figure 3. Graph of Inception Scores for each optimization method varying learning rates.

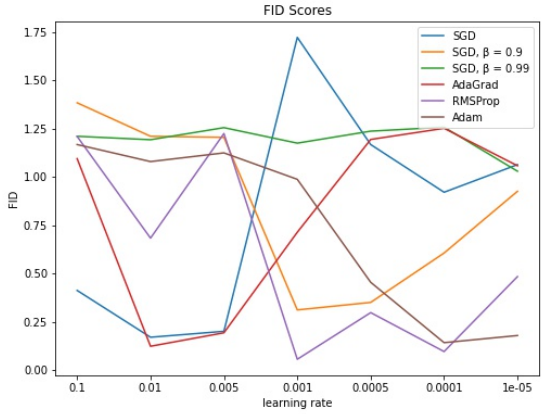


Figure 4. Graph of Fréchet inception distances for each optimization method varying learning rates.

The top 8 IS and their corresponding optimizers are shown in Table 1. Three of the top scores are from models trained using RMSProp as the optimizer. The second highest score was from a model trained using Adam, however no other top IS was from a model trained from Adam. The other top scores are from models that use AdaGrad and SGD as optimizers.

The top IS score for each optimizer is shown in Table 2. We see that the top performing generator was trained with RMProp followed by Adam, AdaGrad, and finally the SGD

Table 1. Top-8 IS scores with optimizer, learning rate, and percent change from SGD with learning rate 0.1 baseline score.

OPTIMIZER	α	IS	% CHANGE
RMSPROP	0.0005	3.7543	46.47%
ADAM	0.0001	3.6794	43.55%
ADAGRAD	0.01	3.5522	38.59%
RMSPROP	0.001	3.4854	35.98%
ADAGRAD	0.005	3.3105	29.16%
RMSPROP	0.0001	3.2442	26.57%
SGD	0.005	3.0543	19.16%
SGD	0.01	2.9883	16.59%

Table 2. Top IS per optimization method with optimizer, learning rate, and percent change from SGD with learning rate 0.1 baseline score.

OPTIMIZER	α	IS	% CHANGE
RMSPROP	0.0005	3.7543	46.47%
ADAM	0.0001	3.6794	43.55%
ADAGRAD	0.01	3.5522	38.59%
SGD	0.005	3.0543	19.16%
SGD, $\beta = 0.9$	0.0005	2.9008	13.18%
SGD, $\beta = 0.99$	0.00001	1.7037	-33.53%

Table 3. Bottom-8 FID scores with optimizer, learning rate, and percent change from SGD with learning rate 0.1 baseline score.

OPTIMIZER	α	FID	% CHANGE
RMSPROP	0.001	0.0572	-86.17%
RMSPROP	0.0001	0.0971	-76.52%
ADAGRAD	0.01	0.1246	-69.87%
ADAM	0.0001	0.1431	-65.40%
SGD	0.01	0.1713	-58.58%
ADAM	0.00001	0.1802	-56.43%
ADAGRAD	0.005	0.1943	-53.02%
SGD	0.005	0.2019	-51.18%

Table 4. Lowest FID per optimization method with optimizer, learning rate, and percent change from SGD with learning rate 0.1 baseline score.

OPTIMIZER	α	FID	% CHANGE
RMSPROP	0.001	0.0572	-86.17%
ADAGRAD	0.01	0.1246	-69.87%
ADAM	0.0001	0.1431	-65.40%
SGD	0.01	0.1713	-58.58%
SGD, $\beta = 0.9$	0.001	0.3127	-24.40%
SGD, $\beta = 0.99$	0.00001	1.0308	149.23%

methods.

The lowest 8 FID and their corresponding optimizers are show in Table 3. The lowest two scores were from models trained using RMSProp. The other optimizers that achieve low FID are AdaGrad, Adam, and SGD.

The lowest FID score for each optimizer is shown in Table 4. We see that the top performing generator was again trained with RMProp followed by Adagrad, Adam, and finally the SGD methods as well.

For SGD, lower learning rates result in poorer images generated, while a learning rate of 0.005 producing our best model trained using SGD. Using a learning rate that is smaller than 0.005 appears to result in overall worse generators, but using higher learning rates also do not appear to produce very good generators.

Adding momentum, learning rates that larger than 0.001 and small than 0.0001 do not appear to do well with a β value of 0.9, however, with a β value of 0.99, only optimization methods with very small learning rates appear result in any kind of image at all. This leads to the conclusion that SGD with momentum in general does not appear to result in well trained generators. Five of the seven models with varying learning rates achieve an IS of 1.000, as seen in Table 6.

For AdaGrad, learning rates that 0.01 and 0.005 achieve fairly good IS and FID out of the experiments done. Again, with learning rates that are too high or too low, the models appear not to achieve any ability to generate feasible images.

For RMSProp, almost all models trained with this optimizer results in a model that achieves higher than the minimum IS. It also has trained the models that produced images with the highest IS and lowest FID out of all trained models. Using RMSProp appears to allow for more learning rates to result in acceptable generators as learning rates from 0.0001 to 0.001 all achieve more than minimum IS and only two of the seven models trained with this optimizer generate images with the minimum IS.

Using Adam, smaller learning rates produce better generators, with the best generator being produced by training with a learning rate of 0.0001. While this produces a fairly high IS and low FID in comparison to other methods, other learning rates with Adam fail to train generators anywhere near the quality of other generators.

There is no optimization method does universally well; all methods result in at least one of the top IS and one of the lower FID (with the exception of SGD with momentum). All optimization methods train models that produce images with an IS of 1.000, the minimum score, for some hyperparameter. This reinforces the idea that hyperparameter is critical in the training of cGANs.

That being said, out of all optimization methods RMSProp appears to have the most success in the experiments run as most models built with varying learning rates result in building generators that produce images with better statistical performance numbers.

4.3. Generated Images

Images generated from the baseline SGD with a learning rate of 0.1 is shown in Figure 5(a). We also include images from the generators that achieve the top three IS,

- RMSProp, $\alpha = 0.0005$ (Figure 5(b)),
- Adam, $\alpha = 0.0001$ (Figure 5(c)),
- AdaGrad, $\alpha = 0.01$ (Figure 5(d)),

lowest three FID,

- RMSProp, $\alpha = 0.001$ (Figure 5(e)),
- RMSProp, $\alpha = 0.0001$ (Figure 5(f)),

and any generators that appear on both the top 8 IS and bottom 8 FID list (and are not already shown)

- AdaGrad, $\alpha = 0.005$ (Figure 5(g)), and
- SGD, $\alpha = 0.005$ (Figure 5(h)).

By visual inspection in Figure 5(a), we see that generator trained by the baseline SGD results in images that don't appear to be distinguishable. Images from one class are not distinguishable from images from another class, neither are they distinguishable as their given class.

The generator with the highest IS was trained using RMSProp ($\alpha = 0.0005$) and its produced images are shown in Figure 5(b). This optimization method produced a generator that creates distinguishable images. Images produced as 'airplane' is clearly distinguishable from images produced as 'apple'. A few of the other classes include images that are not clearly the labelled sketch, notable 'nose' does not appear to generate any great images of noses, however, the images within the class are similar to each other.

The other generators appear to have some success in generating images of certain classes; many of the generators are able to draw an 'apple', for example. However, it appears that more complicated images such as 'bus' still prove to be quite difficult to generate, likely due to the complexity of some of the training data (as seen in Figure 1). Some generators don't appear to produce any kind of distinguishable images, notable in the case of the generator trained with SGD ($\alpha = 0.005$) as seen in Figure 5(h).

Across all generators, there appears to be confusion of classes; for example in the generator trained with RMSProp ($\alpha = 0.001$), images that are generated as 'fish' and 'key' appear to be very similar as seen in Figure 5(e). Additionally, none of the images appear to be very clean; there is some amount of noise in almost every image, unlike in the training data.

5. Discussion

In terms of hardware performance, there appears to be no consensus as to whether one optimizer consistently performs better than another. Each optimizer and hyperparameter produce varying times to train and thus there appear to be no tradeoff in terms of speed in utilizing one optimizer over another. However, because these models were trained on Google Colab in which resources are free, there is some uncertainty involved in what resources are available at any given time [3].

In terms of statistical measures, RMSProp performs the best, producing multiple generators with varying learning rates that outperform the baseline generator. In general, methods with adaptive learning rates (RMSProp, AdaGrad, and Adam) outperform the baseline performance. Because of the importance of synchronized training between the generator and discriminator, the methods that perform smaller updates based on past gradients tend to do well.

SGD with momentum frequently fails to produce generators that compare to the baseline. Because the extra momentum term tends to grant faster convergence, the discriminator often trains far faster than the generator, thus leading to a case in which the discriminator converges and the generator fails to update, producing an overall subpar generator.

Because Adam can be seen as a combination of RMSProp and momentum, we see that Adam performs better than SGD with momentum but worse than RMSProp. Adam might benefit from the adaptive learning rate from RMSProp but take damage from the momentum term.

Because of the varying success within each optimization method, this indicates that hyperparameter optimization is extremely important in the training of these models. As the rate of training for the discriminator and generator must be balanced to avoid one model overpowering another (usually the discriminator overpowering the generator), this tuning of hyperparameters must ensure that the discriminator does not train so fast so that the generator can not train at all.

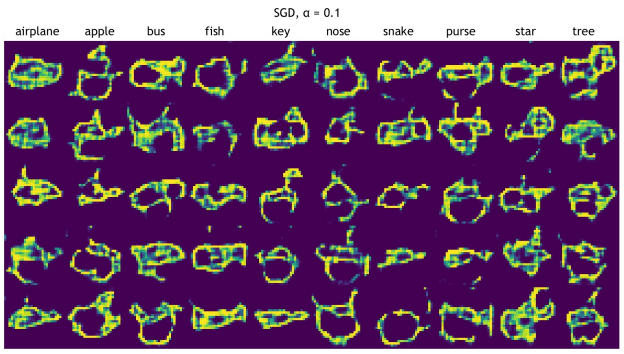
6. Future Work

In order to further improve upon the generation of hand-drawn images from this dataset, there are a few methods to consider.

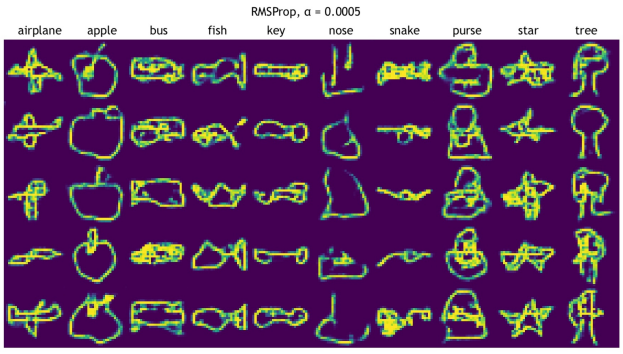
As seen from the experiments, hyperparameters are crucial for the success of an optimization algorithm and different values of certain hyperparameters may work on one optimizer but fail to produce a well trained generator with another optimizer. For this reason, further experiments into hyperparameter optimization could produce better results and further determine what practices should be set in place in utilizing different optimizers.

The experiments in this paper use one model architecture

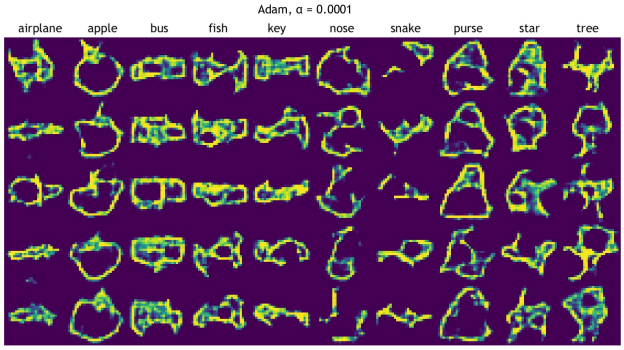
[3] https://research.google.com/colaboratory/faq.html



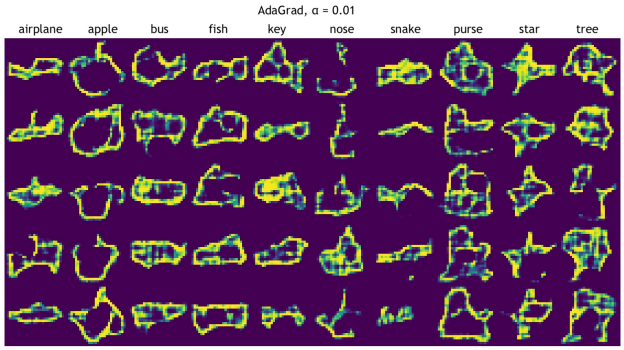
((a)) Images from each class drawn using a cGAN generator trained with SGD ($\alpha = 0.1$).



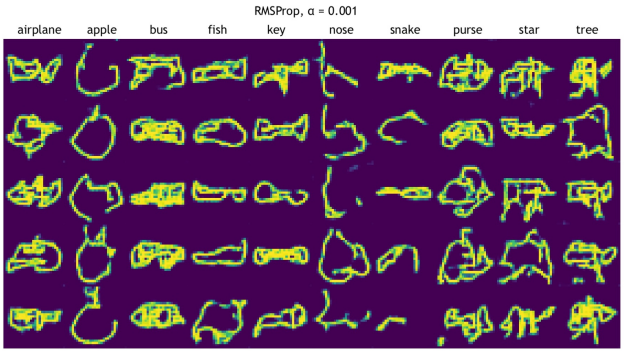
((b)) Images from each class drawn using a cGAN generator trained with RMSProp ($\alpha = 0.0005$).



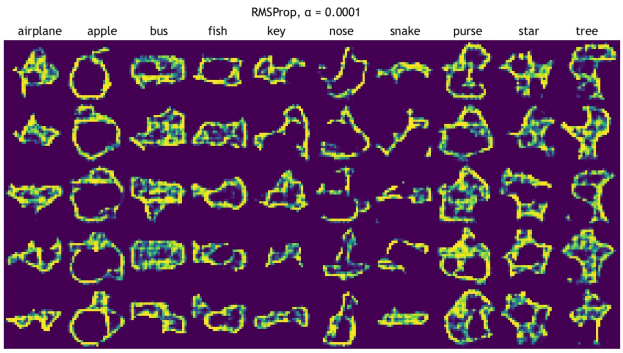
((c)) Images from each class drawn using a cGAN generator trained with Adam ($\alpha = 0.00011$).



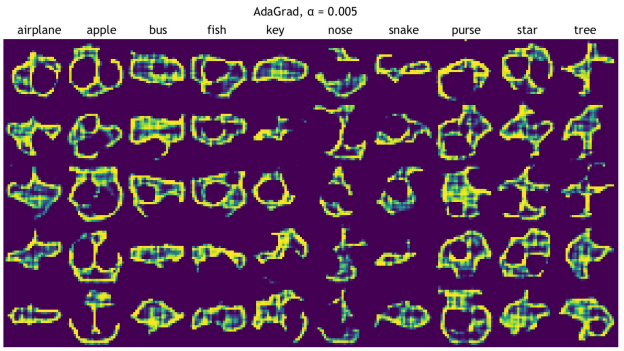
((d)) Images from each class drawn using a cGAN generator trained with AdaGrad ($\alpha = 0.01$).



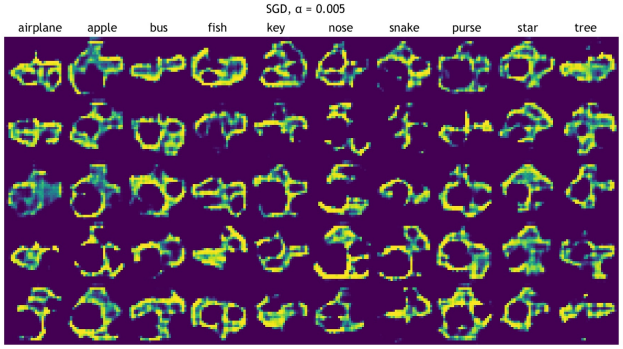
((e)) Images from each class drawn using a cGAN generator trained with RMSProp ($\alpha = 0.001$).



((f)) Images from each class drawn using a cGAN generator trained with RMSProp ($\alpha = 0.0001$).



((g)) Images from each class drawn using a cGAN generator trained with AdaGrad ($\alpha = 0.005$).



((h)) Images from each class drawn using a cGAN generator trained with SGD ($\alpha = 0.005$).

Figure 5. Images from each class drawn using a cGAN generator trained with various optimizers and hyperparameters.

from the Keras tutorial for conditional GANs. While this architecture is shown to produce quite good results for the MNIST dataset (which was used in the example), because of the increasing diversity within examples in Google's Quick, Draw! dataset, more exploration into the model architecture may result in better images.

Finally, because of the fast convergence of the discriminator that renders the generator unable to train, methods to make it more difficult for the discriminator can be used in order to help the generator train. Methods such as instance noise, label smoothing and label noise (Salimans et al., 2016) may help by impeding our discriminator and thus helping our generator train.

7. Conclusion

Training cGANs are a difficult task. It becomes especially difficult with increasing diversity of images such as in hand-drawn sketched images. The optimization method and hyperparameters chosen can greatly affect the training of cGANs and the overall quality of images generated by the generator. There appears to be little tradeoff in choosing one optimizer over another in terms of time to train. While no optimization method dominates in how well images are generated, RMSProp trains models that do well over most hyperparameters, producing more consistent generators that create images with better IS and FID scores. Methods with adaptive learning rates generally do better, allowing both the generator and discriminator to train at the same rate. SGD with momentum trains models that do not do as well, as the momentum term pushes the model to converge faster, allowing the discriminator to converge quickly but impeding the training of the generator.

References

Borji, A. Pros and cons of gan evaluation measures, 2018.

Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., and Bharath, A. A. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65, Jan 2018. ISSN 1053-5888. doi: 10.1109/msp.2017.2765202. URL http://dx.doi.org/10.1109/MSP.2017.2765202.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12 (61):2121–2159, 2011. URL http://jmlr.org/papers/v12/duchi11a.html.

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

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2018.

Karras, T., Aila, T., Laine, S., and Lehtinen, J. Progressive growing of gans for improved quality, stability, and variation, 2018.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization, 2017.

Kodali, N., Abernethy, J., Hays, J., and Kira, Z. On convergence and stability of gans, 2017.

Mirza, M. and Osindero, S. Conditional generative adversarial nets, 2014.

Qian, N. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, 1999. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(98)00116-6. URL https://www.sciencedirect.com/science/article/pii/S0893608098001166.

Ruder, S. An overview of gradient descent optimization algorithms, 2017.

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

Saxena, D. and Cao, J. Generative adversarial networks (gans): Challenges, solutions, and future directions, 2020.

Shahriar, S. Gan computers generate arts? a survey on visual arts, music, and literary text generation using generative adversarial network, 2021.

Table 5. Wall-clock times of model training in seconds.

OPTIMIZER	α	TIME (SEC)
SGD	0.1	1143.6545
SGD	0.01	1270.1910
SGD	0.005	1281.0649
SGD	0.001	1082.2541
SGD	0.0005	1094.4061
SGD	0.0001	1082.0765
SGD	0.00001	1284.9729
SGD, $\beta = 0.9$	0.1	1461.4370
SGD, $\beta = 0.9$	0.01	900.4115
SGD, $\beta = 0.9$	0.005	1462.9218
SGD, $\beta = 0.9$	0.001	1466.2967
SGD, $\beta = 0.9$	0.0005	1283.1591
SGD, $\beta = 0.9$	0.0001	1460.5355
SGD, $\beta = 0.9$	0.00001	1105.1937
SGD, $\beta = 0.99$	0.1	1250.2964
SGD, $\beta = 0.99$	0.01	1076.9800
SGD, $\beta = 0.99$	0.005	1271.8372
SGD, $\beta = 0.99$	0.001	1278.3819
SGD, $\beta = 0.99$	0.0005	1282.5708
SGD, $\beta = 0.99$	0.0001	1460.2150
SGD, $\beta = 0.99$	0.00001	1104.7404
ADAGRAD	0.1	1091.8243
ADAGRAD	0.01	1091.7528
ADAGRAD	0.005	1113.0960
ADAGRAD	0.001	1290.3016
ADAGRAD	0.0005	1473.5040
ADAGRAD	0.0001	1291.7477
ADAGRAD	0.00001	1126.5501
RMSPROP	0.1	1074.6614
RMSPROP	0.01	1458.6742
RMSPROP	0.005	1275.1958
RMSPROP	0.001	1091.1327
RMSPROP	0.0005	1287.8691
RMSPROP	0.0001	1465.1194
RMSPROP	0.00001	1646.3563
ADAM	0.1	1464.9537
ADAM	0.01	1094.6501
ADAM	0.005	1285.4971
ADAM	0.001	1282.0837
ADAM	0.0005	1108.7439
ADAM	0.0001	1098.3483
ADAM	0.00001	1471.2483

Table 6. Final IS and FID scores.

OPTIMIZER	α	IS	FID
SGD	0.1	2.5631	0.4136
SGD	0.01	2.9883	0.1713
SGD	0.005	3.0543	0.2019
SGD	0.001	1.0000	1.7237
SGD	0.0005	1.0000	1.1689
SGD	0.0001	1.0000	0.9219
SGD	0.00001	1.0000	1.0649
SGD, $\beta = 0.9$	0.1	1.1569	1.3850
SGD, $\beta = 0.9$	0.01	1.0000	1.2116
SGD, $\beta = 0.9$	0.005	1.0000	1.2062
SGD, $\beta = 0.9$	0.001	2.7845	0.3127
SGD, $\beta = 0.9$	0.0005	2.9008	0.3515
SGD, $\beta = 0.9$	0.0001	2.2658	0.6073
SGD, $\beta = 0.9$	0.00001	1.0000	0.9269
SGD, $\beta = 0.99$	0.1	NAN	1.2116
SGD, $\beta = 0.99$	0.01	1.0000	1.1938
SGD, $\beta = 0.99$	0.005	1.0000	1.2563
SGD, $\beta = 0.99$	0.001	1.0000	1.1761
SGD, $\beta = 0.99$	0.0005	1.0000	1.2382
SGD, $\beta = 0.99$	0.0001	1.0000	1.2588
SGD, $\beta = 0.99$	0.00001	1.7037	1.0308
ADAGRAD	0.1	1.0000	1.0959
ADAGRAD	0.01	3.5522	0.1246
ADAGRAD	0.005	3.3105	0.1943
ADAGRAD	0.001	2.2567	0.7149
ADAGRAD	0.0005	1.0000	1.1952
ADAGRAD	0.0001	1.0000	1.2529
ADAGRAD	0.00001	1.0000	1.0591
RMSPROP	0.1	1.0000	1.2109
RMSPROP	0.01	2.1374	0.6843
RMSPROP	0.005	1.0000	1.2260
RMSPROP	0.001	3.4854	0.0572
RMSPROP	0.0005	3.7543	0.2988
RMSPROP	0.0001	3.2442	0.0971
RMSPROP	0.00001	2.6036	0.4852
ADAM	0.1	1.0000	1.1689
ADAM	0.01	1.0000	1.0807
ADAM	0.005	1.0000	1.1257
ADAM	0.001	1.1413	0.9888
ADAM	0.0005	2.2878	0.4557
ADAM	0.0001	3.6794	0.1431
ADAM	0.00001	2.4504	0.1802