International Journal of Computer Applications (0975 - 8887)
Volume 176 - No.33, June 2020

Measuring Performance of Generative Adversarial Networks on Devanagari Script

Amogh G. Warkhandkar
Vidyalankar Institute of Technology
Mumbai,
Maharashtra, India

Baasit Sharief
Birla Institute of Technology and Science, Pilani
Hyderabad Campus, Shameerpet,
Telengana, India

Omkar B. Bhambure
Vidyalankar Institute of Technology
Mumbai,
Maharashtra, India

ABSTRACT

The working of neural networks following the adversarial philosophy to create a generative model is a fascinating field. Multiple papers have already explored the architectural aspect and proposed systems with potentially good results however, very few papers are available which implement it on a real-world example. Traditionally, people use the famous MNIST dataset as a Hello, World! example for implementing Generative Adversarial Networks (GAN). Instead of going the standard route of using handwritten digits, this paper uses the Devanagari script which has a more complex structure. As there is no conventional way of judging how well the generative models perform, three additional classifiers were built to judge the output of the GAN model. The following paper is an explanation of what this implementation has achieved.

General Terms

Deep Learning, Neural Networks, Generative Models, Computer Vision, Digital Image Processing

Keywords

Generator, Discriminator, Sequential Models, Denoising, Morphology, Thresholding

1. INTRODUCTION

The Devanagari script, Fig. 1 is an ancient Indian script, with references in Indian culture and history back to the Vedic period. The 47 characters of the script are divided into 14 vowels and 33 consonants. Alike European languages, this script is also written from left to right. It features rounded shapes within squared outlines and a horizontal line that runs along the top of all characters.
In supervised learning, one trains the machine with well-labeled data. This allows for producing output based on previous experience. This implementation maps the input variables to an output variable and uses an algorithm to learn the relationship between them. This involves learning to predict a label associated with the data. The purpose is for the model to generalize to new data. However, in the real world, the convenience of labeled data being available is less likely.
To address this issue, there is a need for networks that can function without labeled data. GANs [4] are unsupervised learning

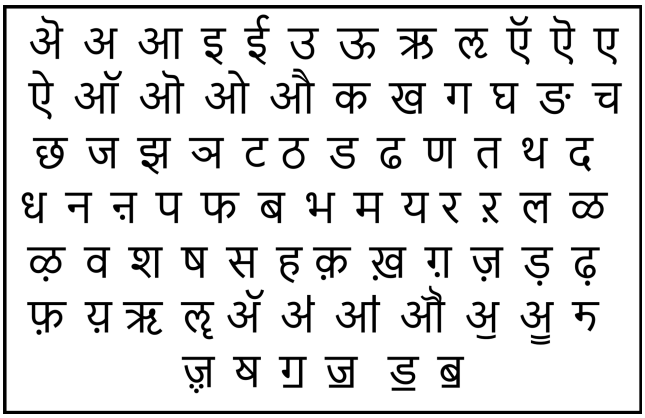

Fig. 1. Devanagari script

algorithms that utilize a supervised loss as part of the training. The data comes in with no labels and there is no attempt to generalize any kind of prediction to new data. The goal is for the GAN to understand what the data looks like with density estimation and generate new examples with what it has understood.
In this implementation [1], Devanagari script characters are fed into the GAN as input. The network will then try to generate the characters as accurately as possible. To calculate the performance metrics of the GAN, the generated characters will be tested on classifiers trained on the original dataset.

2. IMPLEMENTATION

2.1 Generator

The generator model, Fig. 2 creates new images of characters by taking a point from the latent space as input and produces a square grayscale image.
The latent space is an arbitrarily defined vector space of Gaussian-distributed values. In this implementation, the value of latent dim is set to 100. Random points from this space will be drawn and provided to the generator during training. At the end of the training, it will represent a compressed representation of a character.

[1] All code and hyperparameters available at: DevGAN.

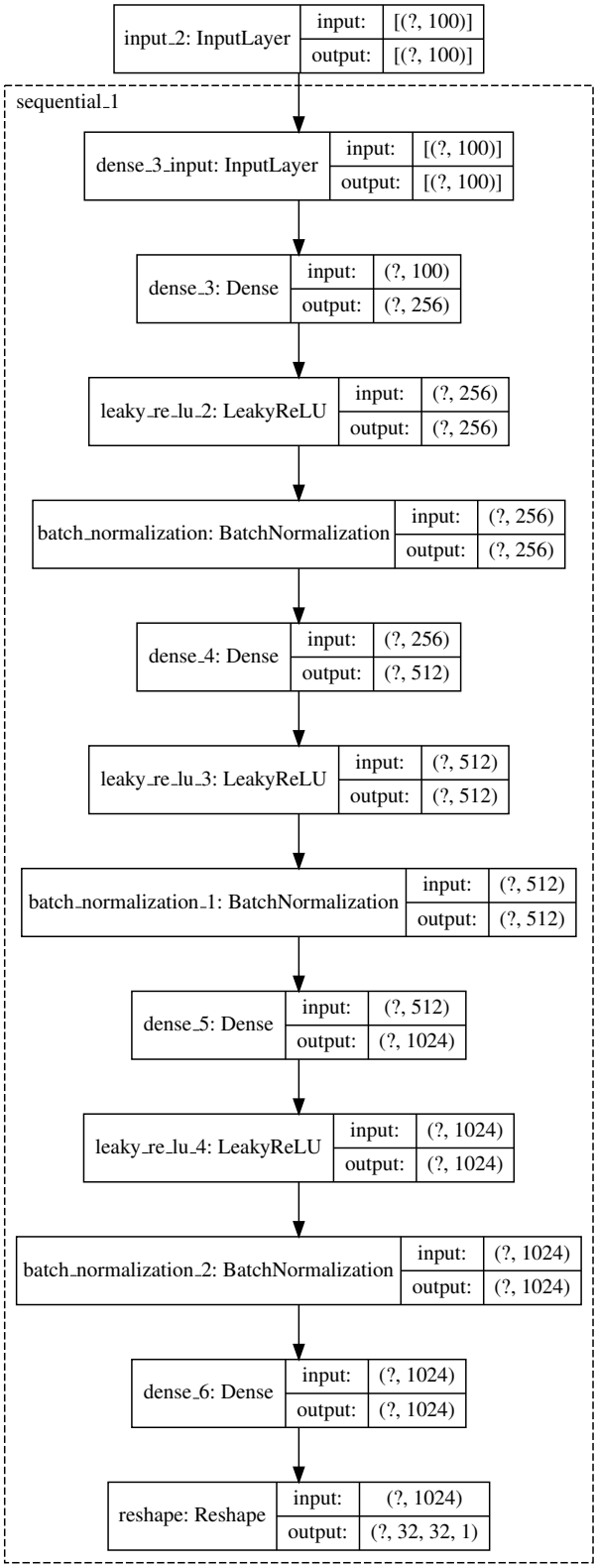

Fig. 2. Architecture of the Generator

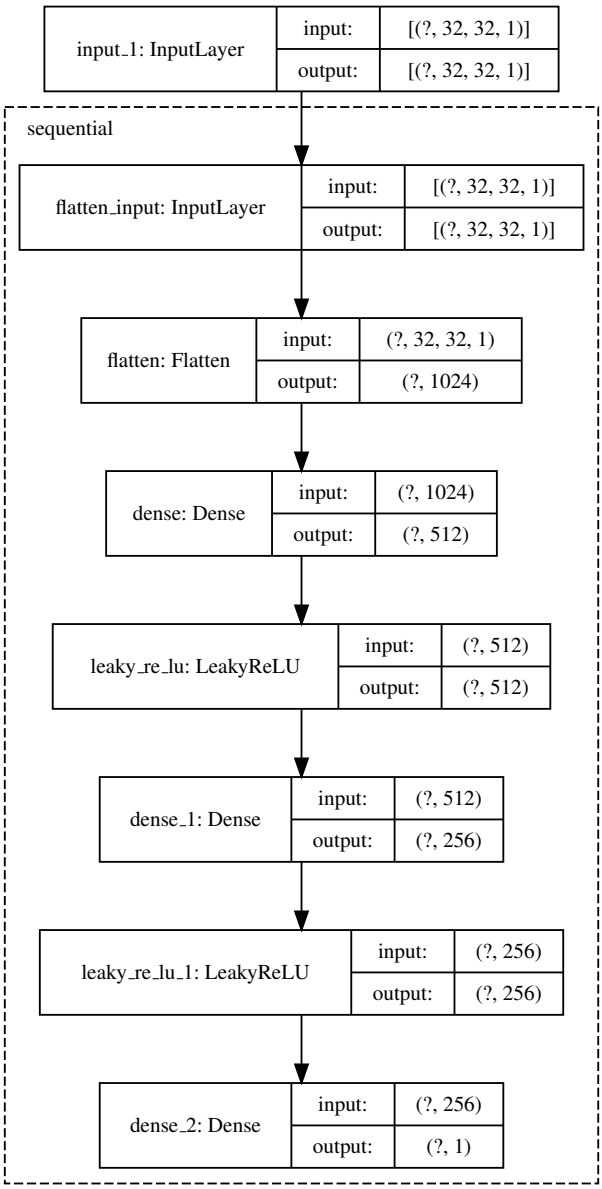

Fig. 3. Architecture of the Discriminator

The architecture consists of three *Dense* layers with the *LeakyReLU* [13] activation function with *alpha* as 0.2 and *BatchNormalization* with *momentum* as 0.8 applied to every *Dense* layer. The final *Dense* layer is equipped with a *Tanh* activation function.

The weights in the generator model are updated based on the performance of the discriminator model. Depending upon the loss output of the discriminator, the rate at which the generator is updated is calculated. Here the adversarial relationship between these two models is defined. The discriminator only concerns itself with the function of distinguishing between real and fake examples. Thus the layers of discriminator are marked as not trainable when combined with the generator model.

The architecture, Fig. 3 consists of a *Flatten* layer and three *Dense* layers with *LeakyReLU* activation function with *alpha* as 0.2 applied to every *Dense* layer. The final *Dense* layer is equipped with *Sigmoid* [8] activation function.

2.2 Classifiers

Classifier 1:
It is a three-layered neural network with two layers consisting of 128 hidden units each. The layers have *ReLU* [2] activation function and the output layer has 10 units with *Softmax* activation function. It uses an *Adam* optimizer with loss function as *sparse categorical crossentropy*.

Classifier 2:
The architecture consists of three *Convolutions* [6][7] and three *Dense* layers. All the Convolutions include *BatchNormalization*, *ReLU* activation function, *MaxPooling*, and *Dropout* [12].
The first *Convolution* consists of 64 *filters* of size (5,5) with *same* padding along with *MaxPooling size* of (2,2) and *strides* of (2,2). The *Dropout* rate is 0.25. The second *Convolution* consists of 32 *filters* of size (3,3) with *valid* padding and *ReLU* activation function and continued with the same *MaxPooling* and same *Dropout* as above. The third *Convolution* consists of 16 *filters* of size (3,3) with *same* padding and *ReLU* activation function and continued with the same *MaxPooling* and same *Dropout*. The Convolutions are continued by three *Dense* layers with the first two of them having *Dropouts* and third being the output layer. The first *Dense* layer consists of 128 units with *ReLU* activation function and *Dropout* with a rate of 0.25. The second *Dense* layer consists of 32 units with *ReLU* activation function and *Dropout* with a rate of 0.5.
The last *Dense* layer has 10 units with *Softmax* activation as the output. The model is compiled with *Adam* [5] optimizer with loss function as categorical crossentropy.

Classifier 3:
This architecture is similar to Classifier 2, with hyperparameters like the number of filters and number of neurons being different and *BatchNormalization* applied after every Dense layer except the output. The model is compiled with *RMSprop* [10] optimizer with the same loss function.

3. OBSERVATIONS

3.1 Dataset

The dataset [1] is an image database [2] of Handwritten Devanagari characters. There are 46 classes of characters with 2000 samples each. The images are in png format with 32x32 resolution. The actual character is centered within 28x28 pixel and a padding of 2 pixels is added on four sides of the character.

Table 1. Classifier Metrics on Original Dataset

	loss	accuracy	val_loss	val_accuracy
Classifier 1	0.0268	0.9920	0.5579	0.8856
Classifier 2	0.2135	0.9415	0.0388	0.9887
Classifier 3	0.0106	0.9964	0.1006	0.9808

[2]Dataset available at: Devanagari Handwritten Character Dataset.

3.2 GAN Output

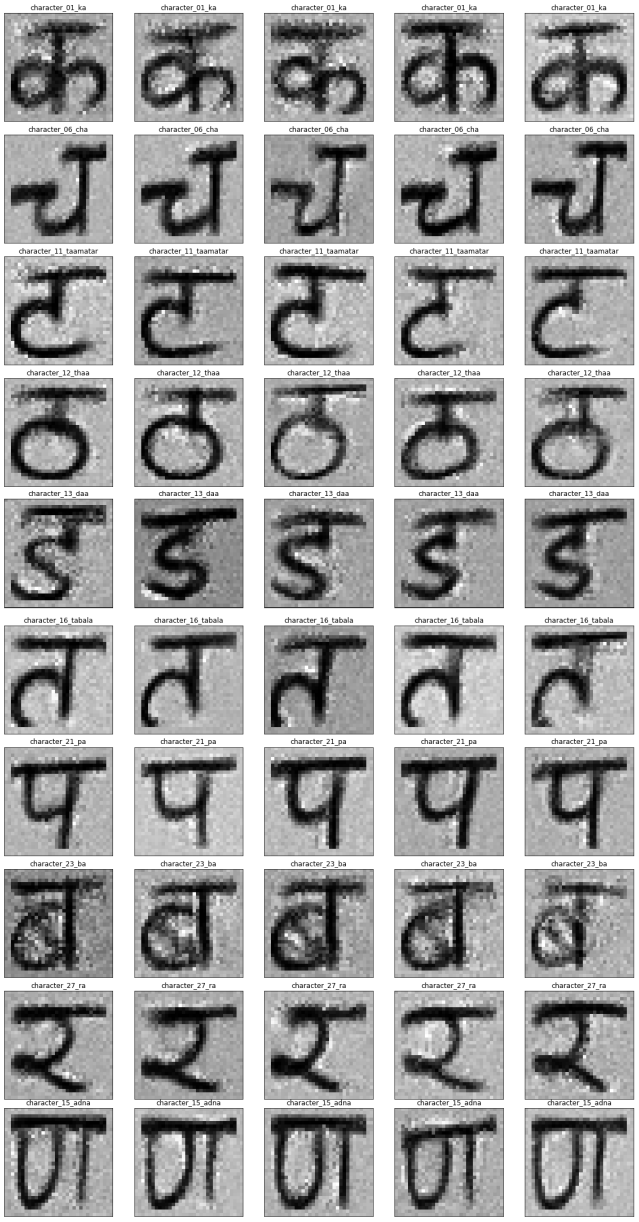

Fig. 4. Generated Characters

The GAN model was trained for 10,000 epochs. At every 500 epoch, the output of the model was stored. Every model started showing promising results at 5000 epochs. Training for 5000 more iterations resulted in better character generation with distinct boundaries between the character and the background. The training time was around 2 hours on an *NVIDIA 1050Ti* GPU. Despite the generated characters being readable to the human eye, Fig 4, a significant amount of noise was observed.

3.3 Classifier Output

All three classifiers performed well on the original dataset, Table 1. However, on the generated characters their performance was very unsatisfactory, Table 2. This failure was accounted to the previously encountered noise in the generated data. Although the generated characters seemed readable to the human eyes, the classifiers were not able to differentiate the characters correctly.

Table 2. Classifier Metrics on Generated Characters

	loss	accuracy
Classifier 1	11.6530	0.0900
Classifier 2	21.9342	0.1230
Classifier 3	8.0340	0.1590

3.4 Generated Data Cleaning

All the images were passed through a *Gaussian Blur* [3] *filter* of size 3x3. Furthermore, *Otsu's Thresholding* [14] was applied to segment the images into *two-pixel* values i.e. 0 and 255. After this, two morphological operations [9][11] were done i.e. opening and closing with *ones* kernel of size 3x3. Finally, they were all passed through a *bitwise NOT* so that the characters resembled the original data, Fig 5.

3.5 Final Classifier Output

After the data cleaning, the result of the classifiers improved greatly. Table 3 contains the statistics and comparisons of the classifier outputs.

Table 3. Classifier Metrics on Cleaned Generated Characters

	loss	accuracy
Classifier 1	1.0490	0.8930
Classifier 2	1.7370	0.8692
Classifier 3	1.2778	0.8660

4. CONCLUSIONS AND FUTURE WORK

Lack of data has always been a big problem in solving real-world challenges involving machine learning and deep learning applications. GANs will be beneficial to a lot of data practitioners because of its possibility of generating real-like artificial data.
This paper has demonstrated the viability of Generative Adversarial Networks on real-life datasets. Even though the generated characters were very noisy, it is safe to assume that in the future there will be a GAN architecture that will take this anomaly into account and create significantly better output. Further experiments can be done on multiple architectures and a hybrid architecture can be developed which might result in the cleaner generation of images. Furthermore, there are high chances of more agile architectures being developed which will make the training and testing faster and more efficient.

5. REFERENCES

[1] S. Acharya, A. K. Pant, and P. K. Gyawali. Deep learning based large scale handwritten devanagari character recognition. In *2015 9th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)*, pages 1–6, 2015.

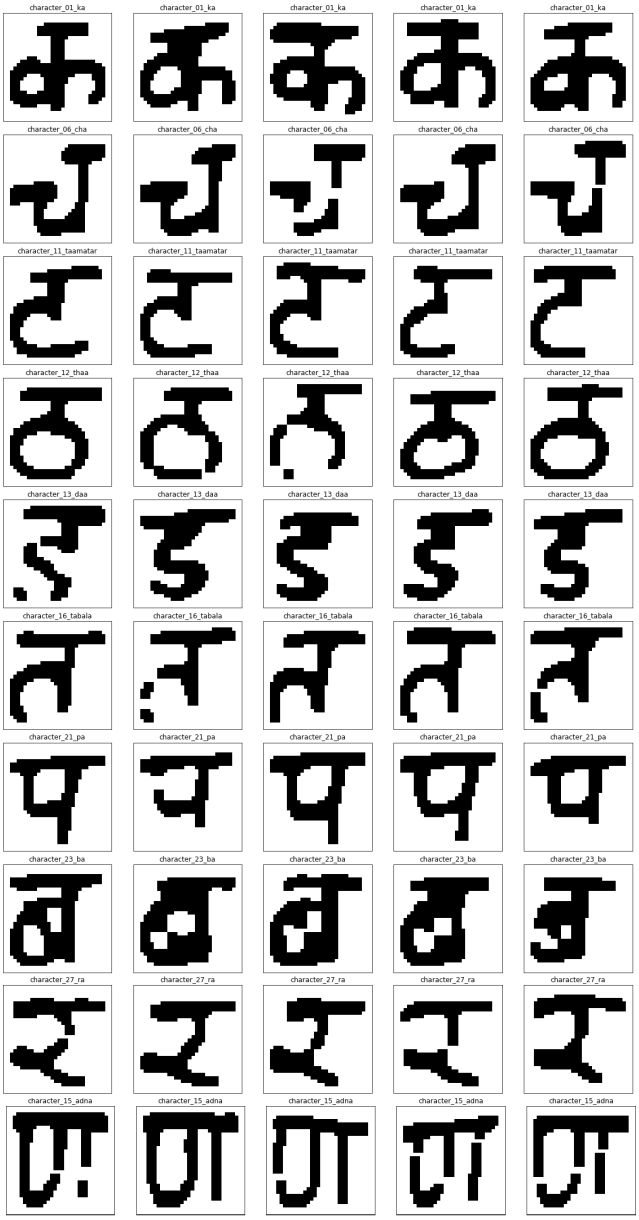

Fig. 5. Cleaned Generated Characters

[2] Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018.

[3] Estevao Gedraite and M. Hadad. Investigation on the effect of a gaussian blur in image filtering and segmentation. pages 393–396, 01 2011.

[4] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[6] Jayanth Koushik. Understanding convolutional neural networks, 2016.

[7] Mengchen Liu, Jiaxin Shi, Zhen Li, Chongxuan Li, Jun Zhu, and Shixia Liu. Towards better analysis of deep convolutional neural networks, 2016.

[8] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018.

[9] A.M Raid, Wael Khedr, Mohamed El-dosuky, and Mona Aoud. Image restoration based on morphological operations. *International Journal of Computer Science, Engineering and Information Technology*, 4:9–21, 07 2014.

[10] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016.

[11] Ravi Srisha and Am Khan. Morphological operations for image processing : Understanding and its applications. 12 2013.

[12] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[13] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015.

[14] Jun Zhang and Jinglu Hu. Image segmentation based on 2d otsu method with histogram analysis. pages 105–108, 01 2008.